IEEE Standards

IEEE Std 1076.4-2000
(Revision of
IEEE Std 1076.4-1995)

IEEE Standard for VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

28 September 2001

Print: SH94959
PDF: SS94959

IEEE Std 1076.4-2000
(Revision of
IEEE Std 1076.4-1995)

IEEE Standard for VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 21 September 2000

IEEE-SA Standards Board

Abstract: The VITAL (VHDL Initiative Towards ASIC Libraries) ASIC Modeling Specification is defined in this standard. This modeling specification defines a methodology which promotes the development of highly accurate, efficient simulation models for ASIC (Application-Specific Integrated Circuit) components in VHDL.
Keywords: ASIC, computer, computer languages, constraints, delay calculation, HDL, modeling, SDF, timing, Verilog, VHDL

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2001 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 28 September 2001. Printed in the United States of America.

Print: ISBN 0-7381-2691-0 SH94959
PDF: ISBN 0-7381-2692-4 SS94959

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

The IEEE and its designees are the sole entities that may authorize the use of IEEE-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

(This introduction is not part of IEEE Std 1076.4-2000, IEEE Standard for VITAL ASIC.)

The objectives of the VITAL (VHDL Initiative Towards ASIC Libraries) initiative can be summed up in one sentence:

Accelerate the development of sign-off quality ASIC macrocell simulation libraries written in VHDL by leveraging existing methodologies of model development.

The VITAL ASIC modeling specification is a revision of the IEEE 1076.4-1995, IEEE Standard for VITAL ASIC Modeling Specification. Several new modeling enhancements have been added to the standard and several usability issues which have been raised with the 1995 standard have been addressed. The new enhancements and usability improvements addressed include:

- Standardized ASIC memory models
- Support of IEEE VHDL93 and SDF 1497 standards
- Multisource interconnect timing simulation
- SKEW constraint timing checks
- Timing constraint checks feature enhancements
- Additional generics to control 'X' generation and message reporting for glitches and timing constraints.
- Negative constraint calculation enhancement for vector signals to support memory models
- Fast delay path disable
- Negative glitch preemption

These new features will improve the functional, timing accuracy significantly and aid performance of gate level VHDL simulations.

The major enhancement is the definition of a ASIC memory modeling standard. With the addition of memory model package VITAL_Memory, a standard is defined which allow memory models to be coded in VHDL more efficiently. The standard VITAL memory model package provides a method to represent memories, procedures and functions to perform various operations and the definition of a modeling style that promotes consistency, maintainability and tool optimization. This standard does not define modeling behavior of specific memories. The scope of the memory model standard is currently restricted to ASIC memory modeling requirement for static RAMs and ROMs. The VITAL memory modeling enhancements are specified in Clause 10 through Clause 12. The VITAL standard memory package is found in Clause 13.

The memory model standard is derived from contributed work from the LSI Logic VHDL behavioral model and Mentor Graphics Memory Table Model (MTM) techniques. The generous support of VHDL International provided the needed funding to take these two contributed works and convert them into the memory specification and package code by the IEEE 1076.4 TAG (Technical Action Group) with significant contribution coming from leading EDA services company GDA Technologies.

The technical direction of the working group as well as the day to day activities of issue analysis and drafting of proposed wordings for the specification are the responsibility of the IEEE 1076.4 TAG. This group consists of Ekambaram Balaji, Prakash Bare, Nitin Chowdhary, Jose De Castro, Martin Gregory, Rama Kowsalya, B. Sudheendra Phani Kumar, William Yam, David Lin, Ashwini Mulgaonkar, Ajayharsh P. Varikat, and Steve Wadsworth and is chaired by Dennis B. Brophy. Without the dedication and hard work of this group it would not have been possible to complete this work.

The VITAL effort germinated from ideas generated at the VHDL International Users' Forum held in Scottsdale, Arizona in May 1992. Further discussions brought people to the conclusion that the biggest impediment to VHDL design was the lack of ASIC libraries; and that the biggest impediment to ASIC library

Copyright © 2001 IEEE. All rights reserved.

development was the lack of a uniform, efficient method for handling timing in VHDL. Since this problem had already been solved for other languages it was clear that a solution in VHDL was possible and that an effective way to arrive at this solution was to leverage existing technology. Leveraging existing tools and environments is viewed as a catalyst for the rapid deployment of ASIC libraries once this initiative is standardized under the IEEE.

The 1076.4 Working Group has a large membership of over three hundred interested people who have made significant contributions to this work through their participation in technical meetings, their review of technical data both in print and through electronic media, and their votes which guided and finally approved the content of the draft specification. This group is chaired by Victor Berman.

The VITAL ASIC modeling specification is the result of numerous discussions with ASIC vendors, EDA tool vendors, and ASIC designers to determine the requirements for effective design and fabrication of ASICS using VHDL. The highest priority issues identified by this group were:

— Timing accuracy
— Model maintainability
— Simulation performance

Some basic guiding principles followed during the entire specification development process were:

— To describe all functionality and timing semantics of the model entirely within the VHDL model and the associated VITAL packages except for multi-source interconnect.
— To provide a set of modeling rules (Level 1) which constrain the use of VHDL to a point that is amenable for simulator optimizations, and at the same time provide enough flexibility to support most existing modeling scenarios.
— To have all timing calculations (load dependent or environmentally dependent) performed outside of the VITAL model. The VITAL model would get these timing values solely as actual values to the model's generic parameter list or via SDF direct import.

Participants

The following persons were members of the 1076.4 Technical Action Group (TAG):

Victor Berman, *Chair*
Dennis B. Brophy, *Chair, 1076.4 Technical Action Group*

Ekambaram Balaji
Prakash Bare
Nitin Chowdhary
Jose De Castro
Martin Gregory
Rama Kowsalya
Phani Kumar
David Lin,
Ashwini Mulgaonkar
B. Sudheendra
Ajayharsh P. Varikat
Steven D. Wadsworth
William Yam

Copyright © 2001 IEEE. All rights reserved.

The following members of the balloting committee voted on this standard:

Peter J. Ashenden
Stephen A. Bailey
David L. Barton
Victor Berman
J Bhasker
William Billowitch
Dennis B. Brophy
Brian A. Dalio
Timothy R. Davis
Ted Elkind
Andrew Guyler
Rich Hatcher
Jim Heaton
Neil G. Jacobson
Osamu Karatsu
Jake Karrfalt
Satoshi Kojima
Gunther Lehmann
Maqsoodul Mannan
Paul J. Menchini
Jean P. Mermet
Quentin G. Schmierer
Steven E. Schulz
Francesco Sforza
Joseph P. Skudlarek
Joseph J. Stanco
Steven D. Wadsworth
Ronald Waxman
Ron Werner
John M. Williams
John Willis
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 21 September 2000, it had the following membership:

Donald N. Heirman, *Chair*
James T. Carlo, *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal
Mark D. Bowman
Gary R. Engmann
Harold E. Epstein
H. Landis Floyd
Jay Forster*
Howard M. Frazier
Ruben D. Garzon
James H. Gurney
Richard J. Holleman
Lowell G. Johnson
Robert J. Kennelly
Joseph L. Koepfinger*
Peter H. Lips
L. Bruce McClung
Daleep C. Mohla
James W. Moore
Robert F. Munzner
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Donald W. Zipse

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Alan Cookson, *NIST Representative*
Donald R. Volzka, *TAB Representative*

Andrew D. Ickowicz
IEEE Standards Project Editor

Copyright © 2001 IEEE. All rights reserved.

Contents

1. Overview 1

1.1 Scope 1
1.2 Purpose 1
1.3 Intent of this standard 1
1.4 Structure and terminology of this standard 1
1.5 Syntactic description 2
1.6 Semantic description 3
1.7 Front matter, examples, figures, notes, and annexes 3

2. References 3

3. Basic elements of the VITAL ASIC modeling specification 4

3.1 VITAL modeling levels and compliance 4
3.2 VITAL standard packages 5
3.3 VITAL specification for timing data insertion 5

4. The Level 0 specification 7

4.1 The VITAL_Level0 attribute 7
4.2 General usage rules 7
4.3 The Level 0 entity interface 8
4.4 The Level 0 architecture body 17

5. Backannotation 19

5.1 Backannotation methods 19
5.2 The VITAL SDF map 20

6. The Level 1 specification 35

6.1 The VITAL_Level1 attribute 35
6.2 The Level 1 architecture body 35
6.3 The Level 1 architecture declarative part 36
6.4 The Level 1 architecture statement part 36

7. Predefined primitives and tables 46

7.1 VITAL logic primitives 46
7.2 VitalResolve 48
7.3 VITAL table primitives 48

8. Timing constraints 54

8.1 Timing check procedures 54
8.2 Modeling negative timing constraints 59

9. Delay selection 70

9.1 VITAL delay types and subtypes 70

Copyright © 2001 IEEE. All rights reserved.

9.2 Transition dependent delay selection ... 71
9.3 Glitch handling ... 71
9.4 Path delay procedures ... 72
9.5 Delay selection in VITAL primitives ... 74
9.6 VitalExtendToFillDelay ... 75

10. The Level 1 Memory specification ... 76

10.1 The VITAL Level 1 Memory attribute ... 76
10.2 The VITAL Level 1 Memory architecture body ... 76
10.3 The VITAL Level 1 Memory architecture declarative part ... 77
10.4 The VITAL Level 1 Memory architecture statement part ... 77

11. VITAL Memory function specification ... 87

11.1 VITAL memory construction ... 87
11.2 VITAL memory table specification ... 90
11.3 VitalDeclareMemory ... 99
11.4 VitalMemoryTable ... 101
11.5 VitalMemoryCrossPorts ... 103
11.6 VitalMemoryViolation ... 105

12. VITAL memory timing specification ... 108

12.1 VITAL memory timing types ... 108
12.2 Memory Output Retain timing behavior ... 109
12.3 VITAL Memory output retain timing specification ... 110
12.4 Transition dependent delay selection ... 110
12.5 VITAL memory path delay procedures ... 111
12.6 VITAL memory timing check procedures ... 116

13. The VITAL standard packages ... 121

13.1 VITAL_Timing package declaration ... 121
13.2 VITAL_Timing package body ... 136
13.3 VITAL_Primitives package declaration ... 163
13.4 VITAL_Primitives package body ... 232
13.5 VITAL_Memory package declaration ... 302
13.6 VITAL_Memory package body ... 323

Annex A (informative) Syntax summary ... 412

Annex B (informative) Glossary ... 418

Annex C (informative) Bibliography ... 420

Copyright © 2001 IEEE. All rights reserved.

IEEE Standard for VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification

1. Overview

This clause describes the purpose and organization of this standard.

1.1 Scope

To provide a standard method of modeling ASICs in VHDL. This method is aimed at providing efficient, accurate, and tool independent simulation suitable for large chip-level designs typical of those which are based on ASICs.

1.2 Purpose

Current industry methods for designing complex chip-level designs rely on proprietary solutions which are based on specific commercial tools. This standard provides an effective means of performing those designs in a standard, non-proprietary manner that is independent of specific tools. This promotes cost effective design flows and promotes healthy levels of competition in the electronic design industry. This standard builds on the work of IEEE 1076 VHDL which is a standard hardware description language designed to allow such tool independent electronic design.

1.3 Intent of this standard

The intent of this standard is to accurately define the Draft Standard VITAL ASIC Modeling Specification. The primary audiences of this standard are the implementors of tools supporting the specification and ASIC modelers.

1.4 Structure and terminology of this standard

This standard is organized into clauses, each of which focuses on some particular area of the definition of the specification. Each page of the formal definition contains ruler-style line numbers in the left margin. Within each clause, individual constructs or concepts are discussed in each subclause.

Copyright © 2001 IEEE. All rights reserved.

Each subclause describing a specific construct or concept begins with an introductory paragraph. If applicable, the syntax of the construct is then described using one or more grammatical productions. A set of paragraphs describing in narrative form the information and rules related to the construct or concept then follows. Finally, each subclause may end with examples, figures, and notes.

1.5 Syntactic description

The form of a VITAL compliant VHDL description is described by means of a context-free syntax, using a simple variant of the Backus Naur Form (BNF); in particular:

a) Lower cased words, some containing embedded underlines, are used to denote syntactic categories, for example:

    VITAL_process_statement

   Whenever the name of a syntactic category is used, apart from the syntax rules themselves, spaces take the place of underlines (thus, "VITAL process statement" would appear in the narrative description when referring to the above syntactic category).

b) Boldface words are used to denote reserved words, for example:

    process

   Reserved words shall be used only in those places indicated by the syntax.

c) A *production* consists of a *left-hand side*, the symbol "::=" (which is read as "can be replaced by"), and a *right-hand side*. The left-hand side of a production is always a syntactic category; the right-hand side is a replacement rule.

   The meaning of a production is a textual-replacement rule: any occurrence of the left-hand side may be replaced by an instance of the right-hand side.

d) A vertical bar separates alternative items on the right-hand side of a production unless it occurs immediately after an opening brace, in which case it stands for itself.

e) Square brackets enclose optional items on the right-hand side of a production.

f) Braces enclose a repeated item or items on the right-hand side of a production. The items may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule.

g) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *unrestricted_variable*_name is syntactically equivalent to name alone.

h) The term simple_name is used for any occurrence of an identifier that already denotes some declared entity.

i) A syntactic category for which no replacement rule is specified is assumed to correspond to the VHDL syntactic category of the same name. In this case the appropriate replacement rule can be found in the IEEE Std 1076-1993 VHDL LRM.

j) A syntactic category beginning with the unitalicized prefix "VITAL_" represents a subset of a VHDL syntactic category.

 Copyright © 2001 IEEE. All rights reserved.

1.6 Semantic description

The meaning of a particular construct or concept and any related restrictions are described with a set of narrative rules immediately following any syntactic productions in the subclause. In these rules, an italicized term indicates the definition of that term, and an identifier appearing in Helvetica font refers to a definition in one of the VHDL or VITAL standard packages or in a VHDL model description. An identifier beginning with the prefix “VITAL” corresponds to a definition in a VITAL standard package.

Use of the words “is” or “shall” in such a narrative indicates mandatory weight. A non-compliant practice may be described as *erroneous* or as an *error*. These terms are used in these semantic descriptions with the following meaning:

erroneous: the condition described represents a non-compliant modeling practice; however, implementations are not required to detect and report this condition. Conditions are deemed erroneous only when it is either very difficult or impossible in general to detect the condition during the processing of a model.

error: the condition described represents a non-compliant modeling practice; implementations are required to detect the condition and report an error to the user of the tool.

1.7 Front matter, examples, figures, notes, and annexes

Prior to this clause are several pieces of introductory material; following the final clause are some annexes and an index. The front matter, annexes, and index serve to orient and otherwise aid the user of this manual but are not part of the definition of the Draft Standard VITAL ASIC Modeling Specification.

Some subclauses of this definition contain examples, figures, and notes; with the exception of figures, these parts always appear at the end of a subclause. Examples are meant to illustrate the possible forms of the construct described. Figures are meant to illustrate the relationship between various constructs or concepts. Notes are meant to emphasize consequences of the rules described in the clause or elsewhere. In order to distinguish notes from the other narrative portions of the definition, notes are set as enumerated paragraphs in a font smaller than the rest of the text. Examples, figures, and notes are not part of the definition of the specification.

2. References

This clause lists the standards upon which this standard depends. Bibliographic references may be found in Annex C. Citations of the form “[C1]” refer to items listed in Annex C, not to items listed in this clause.

IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual.[1]
IEEE Std 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).
IEEE P1497, Draft Standard Delay Format Specification.

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

Copyright © 2001 IEEE. All rights reserved.

3. Basic elements of the VITAL ASIC modeling specification

This standard defines a modeling style for the purpose of facilitating the development and acceleration of sign-off quality ASIC macrocell simulation libraries written in VHDL.

The VITAL ASIC modeling specification is an application of the VHSIC Hardware Description Language (VHDL), described in IEEE Std 1076-1993 [C1][2]. This document uses the term *VHDL* to refer to the VHSIC Hardware Description Language.

This modeling specification relies on the IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_Logic_1164), described in IEEE Std 1164-1993, for its basic logic representation. Throughout this document, the term *standard logic* refers to the Std_Logic_1164 package or to an item declared in the Std_Logic_1164 package.

This modeling specification relies on the IEEE P1497 Draft Standard Delay Format (SDF), as a standard external timing data representation. Throughout this document, the term *SDF* refers to this particular version of the delay format.

The VITAL ASIC modeling specification consists of three basic elements: the formal definition of a VITAL compliant VHDL model, a set of VHDL packages for providing standard timing support, standard functionality support, standard memory functional and timing support, and a semantic specification describing a standard mechanism for insertion of timing data into a VHDL model.

NOTE—VHDL is used to define the VITAL ASIC modeling specification except certain features of multi-source interconnect modeling, direct SDF backannotation and negative constraint calculations. The use of the VITAL package code without specific implementation of these other features does not offer full use of these features.

3.1 VITAL modeling levels and compliance

A VITAL ASIC cell is represented by a VHDL design entity. The VITAL ASIC modeling specification defines the characteristics of a VITAL design entity in terms of the VHDL descriptions of the entity and architecture, and in terms of the associated model which is the result of the elaboration of those VHDL descriptions.

The specification defines three modeling levels; these levels are called *VITAL Level 0*, *VITAL Level 1 and VITAL Level 1 Memory*. Each modeling level is defined by a set of modeling rules. The VITAL Level 0 specification forms a proper subset of the VITAL Level 1 specification and VITAL Level 1 Memory specification. The modeling rules for a VITAL Level 1 specification and VITAL Level 1 Memory specification are mutually exclusive.

A model is said to adhere to the rules in a particular specification only if both the model and its VHDL description satisfy all of the requirements of the specification. Furthermore, if such a model makes use of an item described in a configuration declaration or a package other than a VHDL or VITAL standard package, then the external item shall satisfy the requirements of the specification, as though the item appeared in the VHDL description of the design entity itself.

The VITAL Level 0 specification defines a set of standard modeling rules that facilitate the portability and interoperability of ASIC models, including the specification of timing information. A model which adheres to the rules in the Level 0 specification is said to be a *VITAL Level 0 model*. The Level 0 modeling specification is described in Clause 4.

[2]The numbers in brackets correspond to those of the bibliography in Annex C.

 Copyright © 2001 IEEE. All rights reserved.

The VITAL Level 1 specification defines a usage model for constructing complete cell models in a manner that facilitates optimization of the execution of the models. A model which adheres to the rules in both the Level 0 model interface specification and the Level 1 model architecture specification is said to be a *VITAL Level 1 model*. The Level 1 modeling rules are defined in Clause 6.

The VITAL Level 1 Memory specification defines a usage model for constructing complete memory cell models in a manner that facilitates optimization of the execution of the memory models. A model which adheres to the rules in both the Level 0 model interface specification and the Level 1 Memory model architecture specification is said to be a *VITAL Level 1 Memory model*. The Level 1 Memory modeling rules are defined in clause 10.

A model that is a VITAL Level 0 model or a VITAL Level 1 model or a VITAL Level 1 Memory model is said to be *VITAL compliant*. A VITAL compliant model description contains an attribute specification representing the highest level of compliance intended by the enclosing entity or architecture. Descriptions of these attribute specifications may be found in 4.1, 6.1 and 10.1.

NOTES:

1) A Level 1 model or a Level 1 Memory model is by definition a Level 0 model as well (but not vice versa).

2) The rules outlined in the Level 0, Level 1 and Level 1 Memory specifications apply to model descriptions, not to the VITAL standard packages themselves.

3) AVITAL compliant tool is assumed to enforce the definition of all applicable rules in accordance with the definitions of the terms IS, SHALL, ERROR, and ERRONEOUS. In addition, a compliant tool is expected to accept and correctly execute a VITAL compliant model, and to identify and reject models which are not compliant. A VITAL compliant tool is also expected to fully support the processes described in the specification, including SDF backannotation and negative time sequential constraint transformation.

3.2 VITAL standard packages

The Draft Standard VITAL ASIC Modeling Specification defines three standard packages for use in specifying the timing and functionality of a model: VITAL_Timing, VITAL_Primitives and VITAL_Memory. The text of these packages may be found in Clause 13.

The VITAL_Timing package defines data types and subprograms to support development of macrocell timing models. Included in this package are routines for delay selection, output scheduling, and timing violation checking and reporting.

The VITAL_Primitives package defines a set of commonly used combinatorial primitives and general purpose truth and state tables. The primitives are provided in both function and concurrent procedure form to support both behavioral and structural modeling styles.

The VITAL_Memory package defines data types and subprograms to support development of memory models. Included in this package are routines for functional modeling, corruption handling, memory specific delay selection, output scheduling, and timing violation checking and reporting.

3.3 VITAL specification for timing data insertion

The Draft Standard VITAL ASIC Modeling Specification defines certain semantics that are assumed by a VITAL compliant model and must be implemented by a tool processing or simulating VITAL compliant models which rely on these semantics. These semantics concern the specification and processing of timing data in a VHDL model. They cover SDF mapping, backannotation, and negative constraint processing.

Copyright © 2001 IEEE. All rights reserved.

The timing data for a VITAL compliant model may be specified in Standard Delay Format (SDF). The VITAL SDF Map is a mapping specification that defines the translation between SDF constructs and the corresponding generics in VITAL compliant models. The mapping specification may be used by tools to insert timing information into a VHDL model, either by generating an appropriate configuration declaration, or by performing backannotation through direct SDF import. The VITAL SDF map is defined in 5.2.

The specification introduces two new simulation phases for designs using VITAL models: the backannotation phase, and the negative constraint calculation phase. These phases occur after VHDL elaboration but before initialization.

The backannotation specification defines a backannotation phase of simulation and a mechanism for directly annotating generics with appropriate timing values from SDF (see 5.1.1). The specification also defines the correct state of the timing generics of a model at the end of the backannotation phase.

The negative constraint calculation specification describes a methodology for modeling negative timing constraints in VHDL (see 8.2). It defines a negative constraint calculation phase of simulation and an algorithm for computing and adjusting signal delays, which together transform the negative delays into non-negative ones for the purpose of simulation.

 Copyright © 2001 IEEE. All rights reserved.

4. The Level 0 specification

The Level 0 specification is a set of modeling rules that promotes the portability and interoperability of model descriptions by outlining general standards for VHDL language usage, restricting the form and semantic content of Level 0 design entity descriptions, and standardizing the specification and processing of timing information. General Level 0 modeling rules are defined in this clause, and those relating to the modeling of negative timing constraints are defined in 8.2.1.

4.1 The VITAL_Level0 attribute

A Level 0 entity or architecture is identified by its decoration with the VITAL_Level0 attribute, which indicates an intention to adhere to the Level 0 specification.

VITAL_Level0_attribute_specification ::= attribute_specification

A Level 0 entity or architecture shall contain a specification of the VITAL_Level0 attribute corresponding to the declaration of that attribute in package VITAL_Timing. The entity specification of the decorating attribute specification shall be such that the enclosing entity or architecture inherits the VITAL_Level0 attribute. The expression in the VITAL_Level0 attribute specification shall be the Boolean literal True.

NOTE—Because the required attribute specification representing VITAL compliance indicates the highest level of compliance (see 3.1), a Level 1 architecture or Level 1 Memory architecture, which is also by definition a Level 0 architecture, contains a VITAL_Level1 or VITAL_Level1_Memory attribute specification (see 6.1 and 10.1) rather than a VITAL_Level0 attribute specification. The above rules apply to architectures that are only Level 0.

Example:

attribute VITAL_Level0 **of** VitalCompliantEntity : **entity is** True;

4.2 General usage rules

A Level 0 model shall adhere to general usage rules that address portability and interoperability.

Rules that reference an item declared in a VHDL standard package, a VITAL standard package, or the Std_Logic_1164 package require the use of that particular item. A model description shall not use VHDL scope or visibility rules to declare or use an alternative item with the same name in the place of the item declared in one of these packages.

4.2.1 Standard VHDL usage

A VITAL Level 0 model shall use IEEE Std 1076-1993 features. Use of foreign architecture bodies or package bodies is prohibited.

It is erroneous for a VITAL model to make use of vendor-supplied attributes or other non-normative VHDL constructs such as meta-comments or directives in a manner that affects the function or timing characteristics of the model.

4.2.2 Organization of VITAL compliant descriptions

The VHDL design entity representing a VITAL ASIC cell is described by a pair of design units that reside in one or more VHDL design files. The VITAL ASIC modeling specification imposes no special requirement on the placement of VITAL compliant description within design files, which may contain a mixture of compliant and non-compliant descriptions.

Copyright © 2001 IEEE. All rights reserved.

```
VITAL_design_file ::=
    VITAL_design_unit { VITAL_design_unit }

VITAL_design_unit ::=
      context_clause library_unit
    | context_clause VITAL_library_unit

VITAL_library_unit ::=
      VITAL_Level_0_entity_declaration
    | VITAL_Level_0_architecture_body
    | VITAL_Level_1_architecture_body
    | VITAL_Level_1_memory_architecture_body
```

4.3 The Level 0 entity interface

A VITAL Level 0 entity declaration defines an interface between a VITAL compliant model and its environment.

```
VITAL_Level_0_entity_declaration ::=
    entity identifier is
        VITAL_entity_header
        VITAL_entity_declarative_part
    end [entity] [ entity_simple_name ] ;

VITAL_entity_header ::=
    [ VITAL_entity_generic_clause ]
    [ VITAL_entity_port_clause ]

VITAL_entity_generic_clause ::=
    generic ( VITAL_entity_interface_list ) ;

VITAL_entity_port_clause ::=
    port ( VITAL_entity_interface_list ) ;

VITAL_entity_interface_list ::=
    VITAL_entity_interface_declaration { ; VITAL_entity_interface_declaration }

VITAL_entity_interface_declaration ::=
      interface_constant_declaration
    | VITAL_timing_generic_declaration
    | VITAL_control_generic_declaration
    | VITAL_entity_port_declaration

VITAL_entity_declarative_part ::= VITAL_Level0_attribute_specification
```

The form of this interface strictly limits the use of declarations and statements. The only form of declaration allowed in the entity declarative part is the specification of the VITAL_Level0 attribute. No statement is allowed in the entity statement part.

 Copyright © 2001 IEEE. All rights reserved.

4.3.1 Ports

Certain restrictions apply to the declaration of ports in a VITAL compliant entity interface.

VITAL_entity_port_declaration ::=
    [**signal**] identifier_list : [mode] type_mark [index_constraint] [:= *static*_expression] ;

The identifiers in an entity port declaration shall not contain underscore characters.

A port that is declared in an entity port declaration shall not be of mode LINKAGE.

The type mark in an entity port declaration shall denote a type or subtype that is declared in package Std_Logic_1164. The type mark in the declaration of a scalar port shall denote the subtype Std_Ulogic or a subtype of Std_Ulogic. The type mark in the declaration of an array port shall denote the type Std_Logic_Vector.

NOTE—The syntactic restrictions on the declaration of a port in a Level 0 entity are such that the port cannot be a guarded signal. Furthermore, the declaration cannot impose a range constraint on the port, nor can it alter the resolution of the port from that defined in the standard logic package.

4.3.2 Generics

The generics declared in a Level 0 entity generic clause may be timing generics, control generics, or other generic objects. Timing generics and control generics serve a special purpose in a VITAL compliant model; specific rules govern their declaration and use. Other generics may be defined to control functionality; such generics are not subject to the restrictions imposed on timing or control generics.

4.3.2.1 Timing generics

The VITAL ASIC modeling specification defines a number of *timing generics* which represent specific kinds of timing information. Each kind of timing generic is classified as either a *backannotation timing generic* or a *negative constraint timing generic*, depending on whether the value of the generic is set during the backannotation phase of simulation or the negative constraint calculation phase of simulation. Rules governing the declaration of these generics insure that a mapping can be established between a model's timing generics and corresponding SDF timing information or negative constraint delays.

VITAL_timing_generic_declaration ::=
    [**constant**] identifier_list ::= [**in**] type_mark [index_constraint] [:= *static*_expression] ;

A timing generic is characterized by its name and its type. The naming conventions (see 4.3.2.1.1) communicate the kind of timing information specified as well as the port(s) or delay path(s) to which the timing information applies. The type of a timing generic (see 4.3.2.1.2) indicates which of a variety of forms the associated timing value takes.

A VITAL compliant description may declare any number of timing generics. There are no required timing generics.

Examples:

```
tperiod_Clk  : VITALDelayType    := 5 ns;
tpd_Clk_Q    : VITALDelayType01  := (tr01 => 2 ns, tr10 => 3 ns);
tipd_D       : VITALDelayType01Z := ( 1 ns, 2 ns, 3 ns, 4 ns, 5 ns, 6 ns )
```

Copyright © 2001 IEEE. All rights reserved.

NOTE—The value of a backannotation timing generic is set during the backannotation phase; however, if negative timing constraints are in effect, its value may be adjusted during the subsequent negative constraint calculation phase.

4.3.2.1.1 Timing generic names

The name of a timing generic shall adhere to the naming conventions for timing generics. If the name of a generic does not adhere to these conventions then the generic is not a timing generic.

The form of a timing generic name and its lexical constituents are described by lexical replacement rules similar to the replacement rules for syntactic constructs. White space is included in these rules to enhance readability; however, white space is not permitted within an identifier. Different elements used to construct names are distinguished by enclosing angle brackets which are not themselves part of the name. If a lexical element enclosed by angle brackets does not have a replacement rule then it corresponds to a VHDL identifier described by the text inside the angle brackets. Boldface indicates literal text. Underscores serve as connectors between constituent elements; they are also literal text.

```
<VITALTimingGenericName> ::=
      <VITALBackannotationGenericName>
    | <VITALNegativeConstraintGenericName>

<VITALBackannotationGenericName> ::=
      <VITALPropagationDelayName>
    | <VITALInputSetupTimeName>
    | <VITALInputHoldTimeName>
    | <VITALInputRecoveryTimeName>
    | <VITALInputRemovalTimeName>
    | <VITALInputPeriodName>
    | <VITALPulseWidthName>
    | <VITALInputSkewTimeName>
    | <VITALNoChangeHoldTimeName>
    | <VITALNoChangeSetupTimeName>
    | <VITALInterconnectPathDelayName>
    | <VITALDeviceDelayName>

<VITALNegativeConstraintGenericName> ::=
      <VITALInternalClockDelayName>
    | <VITALInternalSignalDelayName>
    | <VITALBiasedPropagationDelayName>
```

The name of a timing generic is constructed from a timing generic prefix and a number of other elements representing device labels, ports or signals, edges, and conditions. These various elements are combined in a fixed manner, creating three distinct sections of the name: the timing generic prefix, the timing generic port specification, and the timing generic suffix.

A *timing generic prefix* is a lexical element that serves as the beginning of the VHDL simple name of a timing generic. It identifies the kind of timing information that the generic represents, which in turn determines whether the generic is a backannotation timing generic or a negative constraint timing generic. The timing generic prefix consists of the sequence of characters preceding the first underscore in the generic name. It is an error for a model to use a timing generic prefix to begin the simple name of an entity generic that is not a timing generic.

 Copyright © 2001 IEEE. All rights reserved.

The Draft Standard VITAL ASIC Modeling Specification defines the following set of timing generic prefixes:

tpd	tsetup	thold	trecovery	tremoval
tperiod	tpw	tskew	tncsetup	tnchold
tipd	tdevice	ticd	tisd	tbpd

The *timing generic port specification* identifies the port(s) with which the timing data is associated. It may contain both port and instance names. A port that is referenced in a timing generic port specification is said to be *associated* with that timing generic.

The discussion of timing generic names associates timing generics with entity ports; however, a model may use a signal or some other item in place of an entity port. If the port name extracted from a timing generic port specification does not denote a port on the entity, then no assumptions are made about the item denoted by the port name, and no consistency checks are performed between the timing generic and the named item.

Backannotation and negative constraint calculation require the determination of the name(s) of the port(s) associated with a particular timing generic. A port name is *extracted* from the port specification portion of a timing generic name by taking the lexical element corresponding to that port (a sequence of characters that constitute a VHDL identifier, delimited by underscores), as defined by the naming conventions for that sort of a timing generic.

The name of a timing generic may contain a *timing generic suffix* that corresponds to a combination of SDF constructs representing conditions and edges. The forms of these SDF-related suffixes are described by the following rules:

<SDFSimpleConditionAndOrEdge> ::=
    <ConditionName>
  | <Edge>
  | <ConditionName>_<Edge>

<SDFFullConditionAndOrEdge> ::=
  <ConditionNameEdge> [_<SDFSimpleConditionAndOrEdge>]

<ConditionName> ::=
  simple_name

<Edge> ::=
  posedge
 | **negedge**
 | **01**
 | **10**
 | **0z**
 | **z1**
 | **1z**
 | **z0**

<ConditionNameEdge> ::=
    [<ConditionName>_] <Edge>
  | [<ConditionName>_] **noedge**

A condition name is a lexical element which identifies a condition associated with the timing information. The condition name may be mapped to a corresponding condition expression in an SDF file according to the mapping rules described in 5.2.7.3.2.

Copyright © 2001 IEEE. All rights reserved.

An edge identifies an edge associated with the timing information. The edge may be mapped to an edge name specified in an SDF file using the mapping rules described in 5.2.7.3.1.

NOTE—It is assumed that the names in timing generic port specifications will generally denote entity ports; however, a model may instead name other items which may or may not be visible from the enclosing entity declaration (internal signals, for instance). If a port name in a timing generic port specification does not denote a port on the entity then there are no requirements for consistency between the timing generic and the named item (in fact, the named item does not even have to exist), hence no consistency checks are performed. A tool which processes VITAL compliant models may choose to issue a warning in this case.

4.3.2.1.2 Timing generic subtypes

The type mark in the declaration of a timing generic shall denote a VITAL delay type or subtype. These are discussed in 9.1.

If each port name in the port specification of a timing generic name denotes an entity port, the type and constraint of the timing generic shall be consistent with those of the associated port(s). This consistency is defined as follows:

- If the timing generic is associated with a single port and that port is a scalar, the type of the timing generic shall be a scalar form of delay type. If the timing generic is associated with two scalar ports, the type of the timing generic shall be a scalar form of delay type.

- If a timing generic is declared to be of a vector form of delay type, it represents delays associated with one or more vector ports. If such a timing generic is associated with a single port and that port is a vector, the constraint on the generic shall match that on the associated port. If the timing generic is associated with two ports, one or more of which is a vector, the type of the timing generic shall be a vector form of delay type. If the number of scalar subelements in the first port is not equal to the number of scalar subelements in the second port, the length of the index range of the generic shall be equal to the product of the number of scalar subelements in the first port and the number of scalar subelements in the second port. If the number of scalar subelements in the first port is equal to the number of scalar subelements in the second port, the length of the index range of the generic shall be equal to one of the following:

  (a) The product of the number of scalar subelements in the first port and the number of scalar subelements in the second port.

  (b) The number of scalar subelements in the first port.

- If the timing generic is a scalar form of delay type and it is associated with a single port then that port shall be a scalar port. If the timing generic is a scalar form of delay type and it is associated with two ports then both those ports shall be scalar ports.

NOTE—These consistency requirements between timing generics and ports do not apply if the port specification in the timing generic identifies an item that is not an entity port. In this case the model assumes responsibility for the appropriate type and constraint for the timing generic.

4.3.2.1.3 Timing generic specifications

Each form of timing generic represents a particular kind of timing information. Additional restrictions on the name and type or subtype may be imposed on generics representing a particular kind of timing information. A description of the acceptable forms for a particular kind of timing generic is provided in the subclause describing that kind of timing generic.

 Copyright © 2001 IEEE. All rights reserved.

In the following discussion, an *input port* is a VHDL port of mode IN or INOUT. An *output port* is a VHDL port of mode OUT, INOUT, or BUFFER.

4.3.2.1.3.1 Propagation delay

A timing generic beginning with the prefix tpd is a backannotation timing generic representing propagation delay associated with the specified input-to-output delay path. This generic may include an output retain timing specification. Its name is of the form

<VITALPropagationDelayName> ::=
tpd_<InputPort>_<OutputPort> [_<SDFSimpleConditionAndOrEdge>]

The type of a propagation delay generic shall be a VITAL delay type (see 9.1). In the presence of output retain delays the propagation delay generic type shall only be VitalDelayType01Z (VitalDelayArrayType01Z) or VitalDelayType01ZX (VitalDelayArrayType01ZX).

4.3.2.1.3.2 Input setup time

A timing generic beginning with the prefix tsetup is a backannotation timing generic representing the setup time between an input reference port and an input test port. Its name is of the form

<VITALInputSetupTimeName> ::=
tsetup_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of a setup time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.3 Input hold time

A timing generic beginning with the prefix thold is a backannotation timing generic representing the hold time between an input reference signal and an input test signal. Its name is of the form

<VITALInputHoldTimeName> ::=
thold_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of an input hold time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.4 Input recovery time

A timing generic beginning with the prefix trecovery is a backannotation timing generic representing the input recovery time between an input test signal and an input reference signal (similar to a setup constraint). Its name is of the form

<VITALInputRecoveryTimeName> ::=
trecovery_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of an input recovery time generic shall be a simple VITAL delay type (see 9.1).

Copyright © 2001 IEEE. All rights reserved.

4.3.2.1.3.5 Input removal time

A timing generic beginning with the prefix tremoval is a backannotation timing generic representing input removal time between an input test signal and an input reference signal (similar to a hold constraint). Its name is of the form

<VITALInputRemovalTimeName> ::=
tremoval_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

The type of an input removal time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.6 Period

A timing generic beginning with the prefix tperiod is a backannotation timing generic representing the minimum period time. Its name is of the form

<VITALInputPeriodName> ::=
tperiod_<InputPort> [_<SDFSimpleConditionAndOrEdge>]

If present, the edge specifier indicates the edge from which the period is measured.

The type of a period generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.7 Pulse width

A timing generic beginning with the prefix tpw is a backannotation timing generic representing the minimum pulse width. Its name is of the form

<VITALPulseWidthName> ::=
tpw_<InputPort> [_<SDFSimpleConditionAndOrEdge>]

The edge specifier, if present, indicates the edge from which the pulse width is measured. A posedge specification indicates a high pulse, and a negedge specification indicates a low pulse.

The type of a pulse width generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.8 Input skew time

A timing generic beginning with the prefix tskew is a backannotation timing generic representing skew time between a pair of signals. Its name is of the form

<VITALInputSkewTimeName> ::=
tskew_<FirstPort>_<SecondPort> [_<SDFFullConditionAndOrEdge>]

The type of a skew generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.9 No change setup time

A timing generic beginning with the prefix tncsetup is a backannotation timing generic representing the setup time associated with a no change timing constraint. Its name is of the form

 Copyright © 2001 IEEE. All rights reserved.

<VITALNoChangeSetupTimeName> ::=
tncsetup_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

A no change setup time generic shall appear in conjunction with a corresponding no change hold time generic.

The type of a no change setup time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.10 No change hold time

A timing generic beginning with the prefix tnchold is a backannotation timing generic representing a hold time associated with a no change time constraint. Its name is of the form

<VITALNoChangeHoldTimeName> ::=
tnchold_<TestPort>_<ReferencePort> [_<SDFFullConditionAndOrEdge>]

A no change hold time generic shall appear in conjunction with a corresponding no change setup time generic.

The type of a no change hold time generic shall be a simple VITAL delay type (see 9.1).

4.3.2.1.3.11 Interconnect path delay

A timing generic beginning with the prefix tipd is a backannotation timing generic representing the external wire delay to a port, or an interconnect delay which is abstracted as a simple wire delay on the port. Its name is of the form

<VITALInterconnectPathDelayName> ::=
tipd_<InputPort>

The type of an interconnect path delay generic shall be a VITAL delay type (see 9.1).

4.3.2.1.3.12 Device delay

A timing generic beginning with the prefix tdevice is a backannotation generic representing the delay associated with a device (primitive instance) within the cell model. Its name is of the form

<VITALDeviceDelayName> ::=
tdevice_<InstanceName> [_<OutputPort>]

The type of a device delay generic shall be a VITAL delay type (see 9.1).

4.3.2.1.3.13 Internal signal delay

A timing generic beginning with the prefix tisd represents the internal delay for a data or control port and is used to model negative timing constraints (see 8.2). Its name is of the form

<VITALInternalSignalDelayName> ::=
tisd_<InputPort>_<ClockPort>

The type of an internal signal delay generic shall be a VITAL delay type (see 9.1).

Copyright © 2001 IEEE. All rights reserved.

4.3.2.1.3.14 Biased propagation delay

A timing generic beginning with the prefix tbpd represents a propagation delay that is adjusted to accommodate negative timing constraints (see 8.2). Its name is of the form

<VITALBiasedPropagationDelayName> :=
  tbpd_<InputPort>_<OutputPort>_<ClockPort> [_<SDFSimpleConditionAndOrEdge>]

The type of a biased propagation delay generic shall be a VITAL delay type (see 9.1).

There shall exist, in the same entity generic clause, a corresponding propagation delay generic denoting the same ports, condition name, and edge. Furthermore, the type of the biased propagation generic shall be the same as the type of the corresponding propagation delay generic. The size of the biased propagation delay generic shall be equal to the product of the size of the corresponding propopagation delay generic and the size of the clock port.

4.3.2.1.3.15 Internal clock delay

A timing generic beginning with the prefix ticd represents the internal delay for a clock and is used to model negative timing constraints (see 8.2). Its name is of the form

<VITALInternalClockDelayGenericName> ::=
  ticd_<ClockPort>

The type of an internal clock delay generic shall be a VITAL delay type (see 9.1).

The name given for the clock port in an internal clock delay generic name is considered to be a *clock signal name*. It is an error for a clock signal name to appear as one of the following elements in the name of a timing generic:

— As either the input port or output port in the name of a biased propagation delay generic.
— As the input signal name in an internal signal delay timing generic.
— As the test port in a timing check or recovery removal timing generic.

4.3.2.2 Control generics

The VITAL ASIC modeling specification defines a number of *control generics* that provide special support for certain operations.

VITAL_control_generic_declaration ::=
  [**constant**] identifier_list ::= [**in**] type_mark [index_constraint] [:= *static*_expression] ;

A control generic is characterized by a name, a type, and an assumed meaning. Definition and use of these generics is not required; however, if a generic in an entity has a control generic name, then that generic is a control generic, and its declaration shall conform to the rules in effect for that kind of control generic. It is erroneous for a model to use a control generic for other than its stated purpose.

A generic with the name InstancePath shall be of the predefined type String. It is the string representation of the full path name of the current instance.

A generic with the name TimingChecksOn generic shall be of the predefined type Boolean. It may be used to enable timing checks. The value True indicates that timing checks should be enabled.

Copyright © 2001 IEEE. All rights reserved.

The generics XOn, XOnGlitch, and MsgOn, MsgOnGlitch are switches that may be used as standard mechanism for control of 'X' generation and assertion messages relating to glitch violations.

A generic with the name XOn or XOnGlitch shall be of the predefined type Boolean. It may be used to control 'X' generation for path delays. The value TRUE indicates that glitch violations should cause certain output ports to be assigned value 'X'. The value FALSE means that 'X' should not be generated on the output ports for associated glitch violations.

A generic with the name MsgOn or MsgOnGlitch shall be of the predefined type Boolean. It may be used to control assertion messages for path delays. The value TRUE indicates that assertion messages should be issued when glitch violations are encountered. The value FALSE means assertion messages should not be issued.

The generics XOn, XOnChecks, and MsgOn, MsgOnChecks are switches that may be used as standard mechanism for control of 'X' generation and assertion messages relating to timing violations.

A generic with the name XOn or XOnChecks shall be of the predefined type Boolean. It may be used to control 'X' generation for timing checks. The value TRUE indicates that timing violations should cause certain output ports to be assigned value 'X'. The value FALSE means that 'X' should not be generated on the output ports for associated timing violations. The value of the control generic by itself only affects the formal parameter 'Violation' of all timing check procedures in the VITAL standard timing package.

A generic with the name MsgOn or MsgOnChecks shall be of the predefined type Boolean. It may be used to control assertion messages for timing checks. The value TRUE indicates that assertion messages should be issued when timing violations are encountered. The value FALSE means assertion messages should not be issued.

:NOTES:

1) The declaration of a control generic by itself has no effect; the generic must be associated with an appropriate formal parameter of a VITAL standard package subprogram or named in a timing check condition to have the intended effect. Use of a control generic is not limited to these contexts.

2) The XOn and MsgOn generics are similar, but not identical, to the EIA 567A [C3] XGeneration and MGeneration features. In particular, declaration of an XOn or MsgOn generic does NOT automatically enable timing checks.

4.4 The Level 0 architecture body

A VITAL Level 0 architecture body defines the body of a VITAL Level 0 design entity.

VITAL_Level_0_architecture_body ::=
    architecture identifier **of** *entity*_name **is**
        VITAL_Level_0_ architecture_declarative_part
    begin
        architecture_statement_part
    end [architecture] [*architecture*_simple_name] ;

VITAL_Level_0_architecture_declarative_part ::=
    VITAL_Level0_attribute_specification { block_declarative_item }

The entity associated with a Level 0 architecture shall be a VITAL Level 0 entity.

Copyright © 2001 IEEE. All rights reserved.

4.4.1 Timing generic usage

It is an error if the value of a timing generic is read inside a VITAL Level 0 model prior to the initialization phase of simulation.

NOTE—There is no requirement that the usage of a timing generic be consistent with the kind of timing information implied by the generic name.

 Copyright © 2001 IEEE. All rights reserved.

5. Backannotation

The sole point of entry of timing information into a VITAL compliant model is through the timing generics. With the exception of the use of VITAL_Timing and VITAL_Memory routines, all timing calculations are performed outside of the VHDL model, and external timing information is passed to the model through the backannotation timing generics. *Backannotation* is the process of updating the backannotation timing generics with the external timing information. Signal delays which are used to model negative timing constraints are computed in the negative constraint calculation stage of simulation; their calculation is not part of the backannotation process.

The rules governing the backannotation of timing values into a VITAL compliant model and the mapping of SDF constructs to backannotation timing generics define the semantics assumed by models which adhere to the Level 0 specification.

5.1 Backannotation methods

There are two methods for annotating model instances with timing data: through the use of an appropriate VHDL configuration declaration, and through the direct import of timing data from one or more SDF files. An appropriate VHDL configuration declaration can be generated from SDF data, or by some other means. VHDL configuration declarations cannot be used to annotate multi-source interconnect path delays. If a VITAL compliant model derives its timing information from SDF data, then the state of that model at the beginning of simulation shall be the same, regardless of the annotation path employed.

NOTE—It is assumed that an SDF file will be created (possibly by a tool such as a delay calculator) using information which is consistent with the library data (e.g. a VHDL model). This implies that, in general, the data in the SDF file will be consistent with that in a corresponding VITAL compliant model.

5.1.1 Direct SDF import

Direct SDF import is accomplished by reading delay data from one or more SDF files and using this information to directly modify the backannotation timing generics in a VITAL compliant model. The modification of the backannotation timing generics occurs in the *backannotation phase* of simulation, which directly follows elaboration, and directly precedes negative constraint delay calculation. Once the values of the backannotation timing generics have been established and set by the backannotation process, no further modification is permitted except during the negative constraint calculation stage.

The SDF mapping rules are such that an SDF Annotator that performs direct SDF import is responsible for insuring the semantic correctness of the association of delay values with backannotation timing generics. As a consequence, a delay value or a set of delay values must be appropriate for the type class of the corresponding VHDL timing generic, and all applicable VHDL constraints on the value or set of values must be satisfied.

5.1.2 The SDF Annotator

The term *SDF Annotator* refers to any tool in the class of tools capable of performing backannotation from SDF data in a VITAL compliant manner. This class includes tools which generate appropriate configuration declarations from SDF data.

An SDF Annotator shall annotate the backannotation timing generics. Furthermore, it shall report an error upon encountering any form of noncompliance with a requirement of the VITAL ASIC modeling specification related to the SDF mapping or backannotation process. Its behavior after reporting an error is implementation defined.

Copyright © 2001 IEEE. All rights reserved.

Certain SDF constructs are not supported by the VITAL ASIC modeling specification; these constructs are said to be *unsupported*. Unsupported constructs do not result in the modification of backannotation timing generics, nor do they have any other effect on the backannotation process.

If the SDF data fails to provide a value for a backannotation timing generic in a VITAL compliant model, then the value of that timing generic shall not be modified during the backannotation process, and the value which was set during standard VHDL elaboration shall remain in effect.

NOTE—AVITAL SDF Annotator can also annotate generics other than backannotation timing generics (for example, the InstancePath generic). A VITAL SDF Annotator is neither required to annotate nor prohibited from annotating generics on models which are not VITAL compliant.

5.2 The VITAL SDF map

The VITAL SDF Map specifies the mapping between specific SDF constructs and the corresponding VHDL timing generics and their values. Some SDF constructs are mapped directly to specific kinds of timing generics or their timing values, others map to lexical elements which can be used to construct any timing generic name, and others identify items in the VHDL design hierarchy (such as instances or ports) to which timing data is applied.

The name of the corresponding VHDL timing generic is determined according to the rules of the VITAL SDF map. It is an error if there is no translation of a supported SDF construct to a legal VHDL identifier. It is an error if the generic name that the SDF Annotator constructs from the SDF file is not present in the VHDL model.

The following discussion uses portions of the BNF from the Standard Delay Format Specification to describe SDF constructs. An italicized lowercase identifier represents a SDF syntax construct. The definition of a syntax construct is indicated by the symbol ::=, and alternative definitions are separated by the symbol ||=. Keywords appear in boldface. Uppercase identifiers represent variable symbols. The form “item?” represents an optional item. The form “item*” represents zero or more occurrences of the item. The form “item+” indicates one or more occurrences of the item.

5.2.1 Delay file

An SDF delay file consists of a header and a sequence of one or more cells containing timing data.

delay_file ::= (**DELAYFILE** *sdf_header* {*cell*})

The information in each SDF cell of each SDF file is mapped to the corresponding VHDL constructs using general information, such as the time scale, found in the corresponding SDF header.

5.2.2 Header section

The SDF Annotator uses the information in the SDF header to correctly read and interpret the SDF file. In general, the entries in the header section have no direct effect on the backannotation process itself. Some header entries are purely informational, and others (those detailed below) provide information needed by the SDF Annotator to correctly interpret the SDF file.

sdf_header ::= *sdf_version design_name*? *date*? *vendor*? *program_name*?
program_version? *hierarchy_divider*? *voltage*? *process*?
temperature? *time_scale*?

sdf_version ::= (**SDFVERSION** QSTRING)

 Copyright © 2001 IEEE. All rights reserved.

hierarchy_divider ::= (**DIVIDER** HCHAR)

time_scale ::= (**TIMESCALE** TSVALUE)

The *sdf_version* shall refer to version 4.0.

The *hierarchy_divide*r identifies which lexical character (a period or a slash) separates elements of a hierarchical SDF path name.

The *time_scale* determines the time units associated with delay values in the file.

5.2.3 CELL entry

The SDF CELL entry associates a set of timing data with one or more instances in a design hierarchy.

cell ::= (**CELL** *celltype cell_instance* {*timing_spec*})

The timing data in the CELL entry is mapped to a VHDL model as follows:

1) The *cell_instance* and *celltype* constructs are used to locate a path or a set of paths in the VHDL design hierarchy which correspond to the instance(s) to which the data applies.

2) Each supported timing specification in the sequence of *timing_spec* constructs is mapped to the backannotation timing generic(s) of the specified instance(s) in the VHDL design hierarchy, and the corresponding timing data is transformed into value(s) appropriate for the generic(s).

The CORRELATION entry is not supported by the Draft Standard VITAL ASIC Modeling Specification.

5.2.4 INSTANCE and CELLTYPE entries

The SDF cell instance, in conjunction with the SDF cell type, identifies a set of VHDL component instances to which the timing data in a CELL entry applies. This set is constructed by identifying the VHDL component instances (specified by the SDF cell instance) which match the component type (specified by the SDF cell type).

The CELLTYPE entry

celltype ::= (**CELLTYPE** QSTRING)

indicates that the timing data is applicable only to those component instances which correspond to a VHDL component having the name that is specified by the QSTRING variable. Such instances are said to *match* the cell type.

An SDF cell instance is a sequence of one or more INSTANCE entries that name a path or set of paths in the design hierarchy.

cell_instance ::= *instance*
||= (**INSTANCE** [Hierarchical Identifier])
| (**INSTANCE** *)

instance ::= (**INSTANCE** PATH?)

Copyright © 2001 IEEE. All rights reserved.

The first form of cell instance names the path of a particular instance or set of instances. The second form of SDF cell instance is a wildcard which identifies all component instances, in or below the current level of the design hierarchy, which match the cell type.

A VHDL instance described by one or more SDF instance paths is located by mapping each successive path element of the PATH variable of each successive INSTANCE entry to a VHDL block, generate, or component instantiation statement label of the same name at the next level of the design hierarchy, beginning at the level at which the SDF file is applied. Path elements within an SDF PATH IDENTIFIER are separated by hierarchy divider characters. It is an error if, at any level, an appropriate VHDL concurrent statement label does not exist for the corresponding SDF path element. The last path element shall denote a component instance (i.e. it cannot denote a block or generate statement). The VHDL component associated with the instance shall match the cell type.

An SDF path element may contain a bit spec of the form [*integer*] or [*integer*:*integer*]. Such a path element corresponds to one or more expansions of a FOR generate statement. A bit spec containing a single integer corresponds to a single expansion of the generate statement, and a bit spec containing a pair of integers corresponds to a set of expansions described by a range. It is an error if the alphanumeric portion of a path element containing a bit spec does not correspond to the label of a FOR generate statement.

The set of generate statement expansions corresponding to an SDF path element containing a bit spec shall be determined by mapping the SDF integer or pair of integers to the appropriate VHDL index or range. The VHDL value corresponding to a bit spec integer is obtained by mapping the bit spec integer to the VHDL value whose position number is that bit spec integer — base_type’VAL(*integer*), where base_type is the base type of the generate parameter. It is an error if the corresponding VHDL value does not belong to the discrete range specified in the generate statement. The VHDL range corresponding to a pair of integers is constructed by mapping the left and right SDF integers to the corresponding VHDL values representing the left and right bounds, respectively, and then selecting a direction that results in a range that is not a null range.

Example:

The SDF entry

```
(CELL
    (CELLTYPE “DFF”)
    (INSTANCE a1.b1.c1)
    (DELAY
        (ABSOLUTE (IOPATH i1 o1 (10) (20)))
    )
)
```

requires the SDF Annotator to look for the concurrent statement with label a1 at the current level and b1 and c1 at the successive levels below a1. c1 must be the label of a component instantiation statement, and backannotation is performed on the timing generics of c1.

5.2.5 Timing specifications

An SDF timing specification contains delay or timing constraint data that is mapped directly to one or more backannotation timing generics.

timing_spec ::= *del_spec*
||= *tc_spec*

 Copyright © 2001 IEEE. All rights reserved.

An SDF timing specification consists of a data value (or a set of data values) and a number of constructs describing the nature of the data value(s). The constituents of the timing specification are mapped to different, but related, VHDL items. The data value or set of data values is mapped to an appropriate VHDL delay value. The remainder of the timing specification is mapped to a specific timing generic name or pair of names.

5.2.6 Data value mapping

The timing information in an SDF timing specification is specified in terms of *value*, *rvalue,* and *delval_list* constructs. A *value* or *rvalue* can consist of one, two, or three data values corresponding to the minimum, typical and maximum value. However, for annotation to VITAL designs, only one of these values at a time is used. An SDF Annotator shall provide a mechanism for selecting one value from the triple of values.

The type of the timing generic determines the type of the VHDL delay value to which the corresponding SDF timing information is mapped. It is an error if a backannotation timing generic holds fewer delay values than the number specified in the corresponding SDF entry. If a backannotation timing generic is of a transition dependent delay type which contains more values than are specified by the corresponding SDF entry, then the SDF Annotator supplies the remaining delays in the transition dependent delay value according to a predefined mapping.

A simple SDF *value* or *rvalue* is mapped to an equivalent VHDL value of type Time.

An *delval_list* can contain one, two, three, six, or twelve *rvalues* after SDF extension (in which lists of an intermediate length are interpreted as though they had trailing empty parentheses). If the timing generic is of a scalar form of simple delay type, then the corresponding *delval_list* shall contain a single *rvalue*, and the resulting VHDL delay value is a single value of type Time. Otherwise, the timing generic shall be of a scalar form of transition dependent delay type, and the VHDL delay value is constructed by filling each element of the array with the appropriate SDF value, according to the mapping described in Table [C1].

In the presence of RETAIN delay specification the mapping of each element of the array to the corresponding SDF value is described in Table [C2]. Following restriction are placed on the usage of RETAIN delays:

— In the presence of RETAIN delay specification, the number of transition dependant delay values shall be limited to 6 transitions.
— If RETAIN delay specification includes three different rvalues (r1,r2,r3) the corresponding generic type must be VitalDelayType01ZX.
— If RETAIN delays are specified alongwith tristate IOPATH delays, the corresponding generic type must be VitalDelayType01ZX.

In the following table, each row of the table represents a form of SDF *delval_list*, and each column represents the corresponding delay value for a particular transition.

Table 1— Mapping of SDF delay values to VITAL transition dependent delay values

	VITAL transition dependent delay value											
SDF *delval_list*	**tr01**	**tr10**	**tr0z**	**trz1**	**tr1z**	**trz0**	**tr0x**	**trx1**	**tr1x**	**trx0**	**trxz**	**trzx**
v1	v1	v1	v1	v1	v1	v1						
v1 v2	v1	v2	v1	v1	v2	v2						
v1 v2 v3	v1	v2	v3	v1	v3	v2						
v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6						
v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12

Copyright © 2001 IEEE. All rights reserved.

NOTE—An SDF Annotator follows the SDF annotation rules regarding null delay values and extension of lists of rvalues.

Table 2— Mapping of SDF retain delay values to VITAL transition dependent delay values

	VITAL transition dependent delay value											
SDF *retval_list*	tr01	tr10	tr0z	trz1	tr1z	trz0	tr0x	trx1	tr1x	trx0	trxz	trzx
r1 v1	v1	v1	r1	v1	r1	v1						
r1 r2 v1	v1	v1	r1	v1	r2	v1						
r1 r2 r3 v1	v1	v1	v1	v1	v1	v1	r1		r2			r3
r1 v1 v2	v1	v2	r1	v1	r1	v2						
r1 r2 v1 v2	v1	v2	r1	v1	r2	v2						
r1 r2 r3 v1 v2	v1	v2	v1	v1	v2	v2	r1		r2			r3
r1 v1 v2 v3	v1	v2	v3	v1	v3	v2	r1		r1			r1
r1 r2 v1 v2 v3	v1	v2	v3	v1	v3	v2	r1		r2			min (r1, r2)
r1 r2 r3 v1 v2 v3	v1	v2	v3	v1	v3	v2	r1		r2			r3
r1 v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6	r1		r1			r1
r1 r2 v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6	r1		r2			min (r1, r2)
r1 r2 r3 v1 v2 v3 v4 v5 v6	v1	v2	v3	v4	v5	v6	r1		r2			r3
v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12

5.2.7 Mapping to timing generics

The form of the VHDL timing generic name corresponding to an SDF timing specification is determined by the nature of the timing information and by other items, such as ports, that the timing specification references.

The fact that SDF is case sensitive and VHDL is not case sensitive may cause SDF names that differ only in case to map to the same VHDL generic name. Such cases are treated as multiple SDF entries for the same generic.

5.2.7.1 DELAY entry

Different kinds of SDF DELAY entries are mapped to different kinds of backannotation timing generics.

del_spec ::= (**DELAY** {*deltype*})

The delay types PATHPULSE and GLOBALPATHPULSE are not supported by the Draft Standard VITAL ASIC Modeling Specification, nor is the NETDELAY delay definition.

5.2.7.1.1 ABSOLUTE and INCREMENT delay

SDF delay data is designated as incremental or absolute through the form of the delay type construct, *deltype*. The delay type determines how the timing data is applied to the corresponding timing generic(s).

deltype ::= (**ABSOLUTE** *del_def* {*del_def*})
||= (**INCREMENT** *del_def* {del_def})

 Copyright © 2001 IEEE. All rights reserved.

During backannotation, delay values are applied sequentially, in the order that they appear in the SDF file. An absolute delay replaces an existing generic value. An incremental delay value is added to the existing value of the generic.

NOTE—If more than one SDF delay or timing constraint entry maps to the same generic name, the SDF Annotator updates their values using their existing values when appropriate. For example, if the first entry results in updating the value of a certain generic, and a subsequent SDF entry with INCREMENT delays maps to the same generic name, then the new value of the generic is determined by incrementing the previously updated generic value.

5.2.7.1.2 IOPATH delay

The SDF path delay entry can take a simple form or a conditional form.

del_def ::= (**IOPATH** *port_spec port_instance* {*retain_def* } *delval_list*)
||= (**COND** *conditional_port_expr* (**IOPATH** *port_spec port_instance* {*retain_def* } *delval_list*))
retain_def ::= (**RETAIN** *retval_list*)

delval ::=
rvalue
| (*rvalue rvalue*)
| (*rvalue rvalue rvalue*)
delval_list ::=
delval
| *delval delval*
| *delval delval delval*
| *delval delval delval delval*[*delval*] [*delval*]
| *delval delval delval delval delval delval delval* [*delval*] [*delval*] [*delval*] [*delval*] [*delval*]
retval_list ::=
delval
| *delval delval*
| *delval delval delval*

NOTE—Although the SDF specification allows for RETAIN times to be a *delval_list* or *retval_list*, the VITAL Level0 backannotation rules does not allow the usage of *delval_list* for RETAIN times.

Each maps to a propagation delay generic (see 4.3.2.1.3.1) of the form

tpd_<InputPort>_<OutputPort> [_<SDFSimpleConditionAndOrEdge>]

The <InputPort> corresponds to the port name specified in the *port_spec,* and the <OutputPort> corresponds to the port specifed in the *port_path*. The <SDFSimpleConditionAndOrEdge> is derived from the *conditional_port_expr,* if present, and the *port_edge* of the *port_spec*, if present.

Example:

The SDF entry
(IOPATH Input Output (10) (20))
maps to the generic
tpd_Input_Output

Copyright © 2001 IEEE. All rights reserved.

5.2.7.1.3 PORT delay

The SDF port entry

del_def ::= (**PORT** *port_instance* delval_*list)*

maps to an interconnect path delay generic (see 4.3.2.1.3.11) of the form

tipd_<InputPort>

The <InputPort> corresponds to the port specified in the *port_path*.

If the SDF backannotator detects multiple INTERCONNECT delay entries which map to this generic hence indicating multiple sources of this input port, the backannotation of the PORT entry will proceed according to section 5.2.7.1.5.

Example:

The SDF entry
(PORT Input (10) (20))
maps to the generic
tipd_Input

5.2.7.1.4 INTERCONNECT delay

The SDF interconnect delay entry

del_def ::= (**INTERCONNECT** *port_instance port_instance* delval_*list*)

port_instance ::= *port_path*
::= *instance port_path*

maps to an interconnect path delay generic (see 4.3.2.1.3.11) of the form

tipd_<InputPort>

The <InputPort> corresponds to the port specified in the second *port_instance*. The *instance* constructs in this case do not contribute to the corresponding VHDL timing generic name.

If more than one SDF entry maps to the same interconnect path delay generic, then it is assumed that more than one port drives the specified input port. For such cases:

— In the default mode, the SDF Annotator will mark this instance of the interconnect path delay generic as being in multi source mode and store the delay(s) corresponding to this source of the input port in an internal list. The tipd_<Port> generic corresponding to this input port need not be updated in this case. This behavior is described in section 5.2.7.1.5.
— The system can optionally support a mode in which the SDF Annotator shall provide a control to select between the last, the maximum, and the minimum delay values.

 Copyright © 2001 IEEE. All rights reserved.

Example:

The SDF entry
(INTERCONNECT Output Input (10) (20))
maps to the generic
tipd_Input

5.2.7.1.5 Backannotation in multiple source mode

In the multi source mode of operation, during elaboration, an internal array of the following form is created for each instance of the call to VitalWireDelay (in a Level-1 compliant architecture) the port connected to which has multiple sources:

internal array : array(0 to (Number of sources of <Port> - 1)) of <Type>;

Where the interconnect delay from each source is maintained in one location of this array, and where <Type> is one of VitalDelayArrayType, VitalDelayArrayType01 or VitalDelayArrayType01Z depending upon the declaration of the corresponding interconnect path delay generic (tipd_<Port>).

During backannotation, the following rules are applied:

a) On detecting multiple INTERCONNECT entries mapping to the same instance of a interconnect path delay generic, enable multi source mode for this instance of the generic.

b) Create the interal array of delays as shown above with the currently backannotated value of the interconnect path delay generic annotated to each location in this array.

c) Each INTERCONNECT entry (including the current one) updates the location in the internal array corresponding to the source indicated in this entry.

d) Each PORT entry updates all the locations in the internal array.

5.2.7.1.6 DEVICE delays

The SDF device entry can represent the delay on the output of a modeling primitive (used inside an ASIC cell) or the delay on the output of an entire ASIC cell. The VITAL ASIC modeling specification supports only those device entries which specify the delays across primitives used inside a VITAL model. See Clause 7 for a description of the available primitives.

del_def ::= (**DEVICE** [*port_instance] delval_list*)

port_instance ::= *port_path*
::= *instance port_path*

The device entry maps to a device delay generic (see 4.3.2.1.3.12) of the form

tdevice_<InstanceName> [_<OutputPort>]

The <InstanceName> is derived from the name of the SDF instance to which the DEVICE entry applies (it is not derived from the *port_instance* construct). If the SDF instance has a hierarchical name, the lowest level instance name is the <InstanceName>. The optional <OutputPort> is present if *port_instance* is specified.

NOTE—It is expected that the specified <InstanceName> will be the label for a VITAL primitive concurrent procedure call.

Copyright © 2001 IEEE. All rights reserved.

Example:

The SDF entry
(CELL (CELLTYPE “AN2”)
(INSTANCE Top.I1.P1)
(DELAY (ABSOLUTE (DEVICE Z (10) (20))))
)
maps to the generic
tdevice_P1_Z

5.2.7.2 TIMINGCHECK entry

The SDF timing check entry defines a number of timing checks and timing constraints.

tc_spec ::= (**TIMINGCHECK** *tc_def*+)

tc_def ::= *tchk_def*
||= *cns_def*

tchk_def ::= (**SETUP** *port_tchk port_tchk rvalue*)
||= (**HOLD** *port_tchk port_tchk rvalue*)
||= (**SETUPHOLD** *port_tchk port_tchk rvalue rvalue*)
||= (**RECOVERY** *port_tchk port_tchk rvalue*)
||= (**REMOVAL** *port_tchk port_tchk rvalue*)
||= (**RECREM** *port_tchk port_tchk rvalue rvalue*)
||= (**SKEW** *port_tchk port_tchk rvalue*)
||= (**BIDIRECTSKEW** *port_tchk port_tchk rvalue rvalue*)
||= (**WIDTH** *port_tchk value*)
||= (**PERIOD** *port_tchk value*)
||= (**NOCHANGE** *port_tchk port_tchk rvalue rvalue*)

SDF timing constraint entries for forward annotation — *cns_def* entries PATHCONSTRAINT, SUM, DIFF and SKEWCONSTRAINT — are not supported by the VITAL ASIC modeling specification.

Each true timing check definition (i.e. each *tc_def* which is a *tchk_def*) is mapped to one or more VHDL backannotation timing generics. In general, there is a one-to-one correspondence between SDF timing check definitions and VHDL backannotation timing generics. The SDF SETUPHOLD and NOCHANGE constructs are exceptions. Each is processed as though it were replaced by the collectively equivalent setup and hold entries, hence the SDF timing check is mapped to two separate VHDL backannotation timing generics — one each for setup and hold times.

An SDF timing check definition can contain one or two timing check port specifications (*port_tchk*s) which may be further modified with condition and edge constructs. The corresponding VHDL generic name for an SDF timing check is constructed from the timing check type and the timing check port specifications in the following manner:

1) The appropriate generic timing prefix is selected using the following mapping:

SETUP	tsetup	(see 4.3.2.1.3.2)
HOLD	thold	(see 4.3.2.1.3.3)
SETUPHOLD	tsetup	
	thold	
RECOVERY	trecovery	(see 4.3.2.1.3.4)

 Copyright © 2001 IEEE. All rights reserved.

REMOVAL	tremoval	(see 4.3.2.1.3.5)
RECREM	trecovery	
	tremoval	
SKEW	tskew	(see 4.3.2.1.3.8)
BIDIRECTSKEW	tskew	
	tskew	
WIDTH	tpw	(see 4.3.2.1.3.7)
PERIOD	tperiod	(see 4.3.2.1.3.6)
NOCHANGE	tncsetup	(see 4.3.2.1.3.9)
	tnchold	(see 4.3.2.1.3.10)

For BIDIRECTSKEW between two ports, both the corresponding skew generics between the ports shall be updated.

2) The appropriate timing generic port specification is added to the generic name as follows: In the order in which they appear in the SDF entry, the *port* in each *port_tchk* is mapped to the corresponding VHDL timing generic port specification and the result appended to the timing generic name with a preceding underscore.

3) If a *port_tchk* in the timing check definition contains an edge or a condition, then the appropriate timing generic suffix is constructed according to the rules in 5.2.7.2.1, and appended to the timing generic name with a preceding underscore.

5.2.7.2.1 Condition and edge combinations

A *port_tchk* construct in an SDF timing check definition can contain a condition, the *timing_check_condition*, or an edge, in the form of the EDGE_IDENTIFIER in a *port_spec* that is a *port_edge*. A timing check definition can contain one or two *port_tchk* specifications. The conditions and edges associated with these ports are mapped to a timing generic suffix that is appended to the timing generic name with a preceding underscore.

port_tchk ::= *port_spec*
||= (**COND** [qstring] *timing_check_condition port_spec*)

The conditions and edges in a timing check definition that is associated with a single port (i.e. a PERIOD or WIDTH entry) map to the <SDFSimpleConditionAndOrEdge> lexical suffix that is derived from the *timing_check_condition*, if present, and the EDGE_IDENTIFIER of the *port_spec*, if present.

Each of the remaining timing check definitions is associated with a pair of ports, and the conditions and edges map to the <SDFFullConditionAndOrEdge> lexical suffix

<ConditionNameEdge> [_<SDFSimpleConditionAndOrEdge>]

The <ConditionNameEdge> portion is derived from the first *port_tchk* construct. If the first *port_tchk* does not have an edge, then the <ConditionNameEdge> is of the form

[<ConditionName>**_**] **noedge**

Otherwise, the <ConditionNameEdge> is derived from the *timing_check_condition*, if present, and the EDGE_IDENTIFIER of the *port_spec,* if present.

The <SDFSimpleConditionAndOrEdge> is derived from the second *port_tchk* construct, using the *timing_check_condition*, if present, and the EDGE_IDENTIFIER of the *port_spec*, if present.

Copyright © 2001 IEEE. All rights reserved.

Examples:

The SDF entry
(COND RESET == 1'b1 && CLK == 1'b1 (IOPATH posedge A Y (10) (20)))
maps to the VHDL identifier
tpd_A_Y_RESET_EQ_1_AN_CLK_EQ_1_posedge

The SDF entry
(SETUP (COND Reset == 1'b1 DATA) (posedge CLK) (5))
maps to the VHDL identifier
tsetup_DATA_CLK_RESET_EQ_1_noedge_posedge

5.2.7.3 Mapping of SDF constructs to general VHDL lexical elements

Certain SDF constructs are not themselves mapped to a specific kind of timing generic; instead, they are mapped to lexical elements which may be used to construct any backannotation timing generic name. These general SDF constructs include edges, conditions, and port specifications.

5.2.7.3.1 Edges

The SDF edge construct maps to a VHDL lexical suffix that is textually equivalent to the EDGE_IDENTIFIER.

EDGE_IDENTIFIER ::= **posedge**
||= **negedge**
||= **01**
||= **10**
||= **0z**
||= **z1**
||= **1z**
||= **z0**

This lexical suffix is attached to the VHDL generic identifier with an underscore. The location of the lexical suffix within the generic identifier is determined by the context of the edge.

5.2.7.3.2 Conditions

The SDF conditional construct — identified by the keyword COND — can appear in two different contexts: in a conditional path delay of the form

(**COND** *conditional_port_expr* (**IOPATH** *port_spec port_path delval_list*))

and as a *port_tchk* specification in a timing check definition

(**COND** *timing_check_condition port_spec*)

In either case, the condition is mapped to a a legal VHDL lexical representation of the conditional expression that is then used to construct a suffix for a timing generic name.

The VHDL lexical suffix is constructed from the conditional expression (the *conditional_port_expr* or *timing_check_condition*) using the following algorithm:

1) Separate each part of the condition with an underscore, removing any white space

 Copyright © 2001 IEEE. All rights reserved.

2) Replace each SCALAR_CONSTANT symbol as follows:

1'b0, 1'B0, 'b0, 'B0, 0 by 0

1'b1, 1'B1, 'b1, 'B1, 1 by 1

3) Replace each operator symbol as follows:

(by OP
) by CP
{ by OB
} by CB
[by OSB
] by CSB
, by CM
? by QM
: by CLN (only in expr ? expr : expr statements)
+ by PL
- by MI
* by MU
/ by DI
% by MOD
== by EQ
!= by NE
=== by EQ3
!== by NE3
&& by AN
|| by OR
< by LT
<= by LE
> by GT
>= by GE
& by ANB
| by ORB
^ by XOB
^~ by XNB
~^ by XNB
>> by RS
<< by LS
! by NT
~ by NTB
~& by NA
~| by NO

4) Replace each range as follows:

[x:y] by xTOy
[x] by x
where x and y represent indices

Copyright © 2001 IEEE. All rights reserved.

Example:

The SDF entry
(COND RESET == 1'b1 && CLK == 1'b1 (IOPATH A Y (10) (20)))
is mapped to the VHDL generic
tpd_A_Y_RESET_EQ_1_AN_CLK_EQ_1

5.2.7.3.3 Ports

The SDF *port_spec* construct names a port, and possibly an edge, associated with a timing value. An edge, if present, is processed separately, and is not necessarily adjacent to the corresponding port name in the resulting VHDL identifier. Processing of edges is discussed in the appropriate contexts.

```
port_spec     ::=  port_path
              ||=  port_edge

port_instance ::= port
              ||=  PATH HCHAR port

port_edge     ::=  ( EDGE_IDENTIFIER port_path )

port     ::= scalar_port
         ||= bus_port

scalar_port  ::= IDENTIFIER
             ||= IDENTIFIER [ DNUMBER ]

bus_port     ::= IDENTIFIER [ DNUMBER : DNUMBER ]
```

The form of the SDF port name may impose certain requirements on the subtype of the corresponding VHDL generic. For backannotation purposes, the hierarchical form of a *port_path* is equivalent to its simple form, *port*.

The IDENTIFIER in a *port* maps to a VHDL name which is textually equivalent to the IDENTIFIER. The result shall be a legal VHDL identifier.

If the *port* is of one of the forms

IDENTIFIER [DNUMBER]
IDENTIFIER [DNUMBER : DNUMBER]

then it is assumed that the corresponding VHDL port is a vector, and the associated timing generic shall be of a vector form of VITAL delay type. It is an error if a backannotation timing generic is of a vector form of delay type and none of the corresponding ports in the SDF file have an index or range specification.

Each *port* entry with an index or range specification maps to an element or set of elements in the corresponding timing generic array value. Each such element is denoted by an index which is derived from the SDF index or range specification according to the rules in the following subclauses. It is an error if the array index of the generic element corresponding to a particular SDF entry is out of range for that generic.

NOTE—An escape character in an SDF IDENTIFIER is an error because it cannot be mapped to a legal VHDL identifier.

 Copyright © 2001 IEEE. All rights reserved.

5.2.7.3.3.1 Mapping of a single bus port

The scheme outlined in this subclause applies to an SDF entry which has a single port specification that has an index or range specification. This scheme also applies to an SDF entry which has two port specifications, only one of which has an index or range specification. For any port having an index specification, let

c	denote the index of the SDF port
$j1, j2$	denote the left and right indices of the corresponding port in the VHDL model
$g1, g2$	denote the left and right indices of the corresponding generic in the VHDL model

For the SDF port index c, the SDF Annotator computes the corresponding index x in the generic array by using the following row-dominant scheme:

$$x = g2 + \text{abs}(c - j2)*(g1 - g2)/\text{abs}(g1 - g2)$$

If the SDF port has a range specification rather than an index specification then the above computation is performed for each delay value in the range of the SDF port.

NOTES:

1) For a generic with a descending range constraint of (g1 downto g2), the index computation reduces to

x = g2 + abs(c - j2)

2) For a generic with an ascending range constraint of (g1 to g2), the index computation reduces to

x = g2 - abs(c - j2)

5.2.7.3.3.2 Mapping of two bus ports

The scheme outlined in this subclause applies to an SDF entry that has two port specifications, both of which have an index or range specification. Let

r, c	denote the index of the respective ports in the SDF entry
$i1, i2$	denote the left and right indices of the VHDL port corresponding to the first SDF port
$j1, j2$	denote the left and right indices of the VHDL port corresponding to the second SDF port
$g1, g2$	denote the left and right indices of the corresponding generic in the VHDL model

For the SDF entry having index r and c as the indices for the first and second ports, the SDF Annotator computes the corresponding index x in the generic array by using the following row-dominant scheme:

If the generic is a timing generic, where the number of scalar subelements in the first and second ports are equal, and the generic size is equal to the number of scalar subelements in the first port, the index is computed as follows :

$$x = g2 + \text{abs}(c - j2)*(g1 - g2)/\text{abs}(g1 - g2)$$

It is an error if abs(r - $i2$) != abs(c - $j2$).

Otherwise, the index is computed as follows:

$$x = g2 + (\text{abs}(c - j2) + \text{abs}(r - i2)*(\text{abs}(j1 - j2) + 1))*(g1 - g2)/\text{abs}(g1 - g2)$$

Copyright © 2001 IEEE. All rights reserved.

If one or more of the SDF ports has a range specification rather than an index specification then the above computation is performed for each delay value in the range of the SDF ports.

NOTES:

1) For a generic with a descending range constraint of (g1 downto g2), the index computation reduces to

x = g2 + abs(c - j2)

or

x = g2 + abs(c - j2) + abs(r - i2) * (abs(j1 - j2) + 1)

2) For a generic with an ascending range constraint of (g1 to g2), the index computation reduces to

x = g2 - abs(c - j2)

or

x = g2 - abs(c - j2) - abs(r - i2) * (abs(j1 - j2) + 1)

Example:

Assuming the following VHDL declarations

```
generic (tpd_A_Y : VitalDelayArrayType01 (0 to 3) := (others => (0 ns, 0 ns)));
port (A : IN std_logic_vector (0 to 1);
      Y : OUT std_logic_vector (1 to 2));
```

the SDF entry

```
(IOPATH    A[0]     Y[1] (10) (20))
```

will cause the SDF Annotator to annotate the delay value (10, 20) onto the generic subelement tpd_A_Y(0), and the SDF entry

```
(IOPATH    A[0:1]   Y[1:2] (10) (20))
```

will cause the SDF Annotator to annotate the delay value (10, 20) onto the generic subelements tpd_A_Y(0), tpd_A_Y(1), tpd_A_Y(2) and tpd_A_Y(3).

Assuming the following VHDL declarations

```
generic (tpd_DATAIN_DATAOUT : VitalDelayArray01 ( 0 to 3 ) := (others => (0 ns, 0 ns)));
port ( DATAIN : IN std_logic_vector( 0 to 3 );
       DATAOUT : OUT std_logic_vector( 1 to 4 ) );
```

the SDF entry

```
(IOPATH    DATAIN[0]  DATAOUT[1] (10) (20))
```

will cause the SDF Annotator to annotate the delay value (10, 20) onto the generic subelement tpd_DATAIN_DATAOUT(0).

 Copyright © 2001 IEEE. All rights reserved.

6. The Level 1 specification

The Level 1 specification is a set of modeling rules which constrains the descriptions of cell models in order to facilitate the optimization of the set-up and execution of the models, leading to higher levels of performance than could be expected through the acceleration of the basic capabilities provided by the VITAL standard packages alone.

A Level 1 model description defines an ASIC cell in terms of functionality, wire delay propagation, timing constraints, and output delay selection and scheduling.

6.1 The VITAL_Level1 attribute

A Level 1 architecture is identified by its decoration with the VITAL_Level1 attribute, which indicates an intention to adhere to the Level 1 specification.

VITAL_Level1_attribute_specification ::= attribute_specification

A Level 1 architecture shall contain a specification of the VITAL_Level1 attribute corresponding to the declaration of that attribute in package VITAL_Timing. The expression in the VITAL_Level1 attribute specification shall be the Boolean literal True.

Example:

```
attribute VITAL_Level1 of VitalCompliantArchitecture : architecture is True;
```

6.2 The Level 1 architecture body

A VITAL Level 1 architecture body defines the body of a VITAL Level 1 design entity.

VITAL_Level_1_architecture_body ::=
architecture identifier **of** *entity*_name **is**
VITAL_Level_1_architecture_declarative_part
begin
VITAL_Level_1_architecture_statement_part
end [*architecture*_simple_name] ;

A VITAL Level 1 architecture shall adhere to the Level 0 specification, except for the declaration of the VITAL_Level0 attribute.

The entity associated with a Level 1 architecture shall be a VITAL Level 0 entity. Together, these design units comprise a *Level 1 design entity*.

The only signals that shall be referenced in a Level 1 design entity are entity ports and internal signals. References to global signals, shared variables and signal-valued attributes are not allowed. Each signal declared in a Level 1 design entity shall have at most one driver.

The use of subprogram calls and operators in a Level 1 architecture is limited. The only operators or subprograms that shall be invoked are those declared in package Standard, package Std_Logic_1164, or the VITAL standard packages. Formal subelement associations and type conversions are prohibited in the associations of a subprogram call

References to deferred constants are not allowed.

Copyright © 2001 IEEE. All rights reserved.

6.3 The Level 1 architecture declarative part

The Level 1 architecture declarative part contains declarations of items that are available for use within the Level 1 architecture.

```
VITAL_Level_1_architecture_declarative_part ::=
    VITAL_Level1_attribute_specification
    { VITAL_Level_1_block_declarative_item }

VITAL_Level_1_block_declarative_item ::=
      constant_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | VITAL_internal_signal_declaration
```

6.3.1 VITAL internal signals

A signal that is declared in the declarative part of an architecture is an *internal signal*.

VITAL_internal_signal_declaration ::=
    signal identifier_list : type_mark [index_constraint] [:= expression] ;

The type mark in the declaration of an internal signal shall denote the standard logic type Std_Ulogic or Std_Logic_Vector.

6.4 The Level 1 architecture statement part

The statement part of a VITAL Level 1 architecture is a set of one or more concurrent statements that perform specific VITAL activities.

```
VITAL_Level_1_architecture_statement_part ::=
    VITAL_Level_1_concurrent_statement { VITAL_Level_1_concurrent_statement }

VITAL_Level_1_ concurrent_statement ::=
      VITAL_wire_delay_block_statement
    | VITAL_negative_constraint_block_statement
    | VITAL_process_statement
    | VITAL_primitive_concurrent_procedure_call
```

A Level 1 architecture shall contain at most one wire delay block statement.

If the entity associated with a Level 1 architecture declares one or more timing generics representing internal clock or internal signal delay then negative constraints are in effect, and the Level 1 architecture shall contain exactly one negative constraint block to compute the associated signal delays.

A Level 1 architecture shall contain at least one VITAL process statement or VITAL primitive concurrent procedure call.

Copyright © 2001 IEEE. All rights reserved.

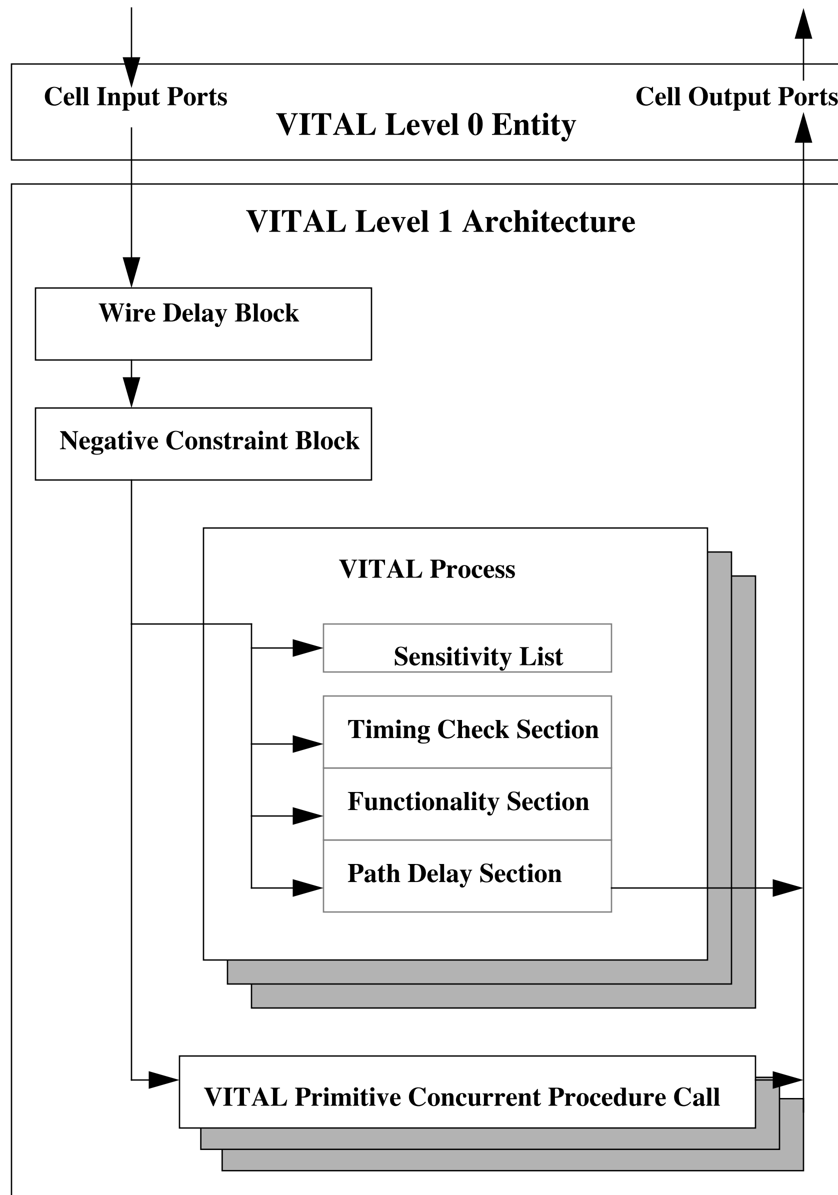


Figure 1—Structure of a VITAL Level 1 model

6.4.1 Wire delay block statement

Interconnect delay between ASIC cells is modeled as an approximation of wire delays at input ports. Wire delays external to a model are propagated inside the model through a *VITAL wire delay block*.

VITAL_wire_delay_block_statement ::=
    *block*_label :
    block
    begin
        VITAL_wire_delay_block_statement_part
    end block [*block*_label] ;

Copyright © 2001 IEEE. All rights reserved.

VITAL_wire_delay_block_statement_part ::=
{ VITAL_wire_delay_concurrent_procedure_call
| VITAL_wire_delay_generate_statement }

VITAL_wire_delay_generate_statement ::=
*generate*_label :
for VITAL_wire_delay_generate_parameter_specification **generate**
{ VITAL_wire_delay_concurrent_procedure_call }
end generate [*generate*_label] ;

VITAL_wire_delay_generate_parameter_specification ::=
identifier **in** *range*_attribute_name

VITAL_wire_delay_concurrent_procedure_call ::= concurrent_procedure_call

The label of a VITAL wire delay block shall be the name WireDelay.

A wire delay is modeled by a concurrent procedure call which invokes one of the VitalWireDelay procedures that are declared in the package VITAL_Timing. A VitalWireDelay procedure delays an input signal by a specified delay value using a transport delay. A wire delay block is the only context in which a call to a VitalWireDelay procedure is allowed.

A port that is associated with a wire delay concurrent procedure call is said to have an *associated wire delay*. A wire delay block shall contain at most one wire delay for each port of mode IN or INOUT declared in the VITAL Level 1 design entity.

Associated with each external wire delay is an internal signal representing the delayed port; this internal signal is called the *wire delayed signal*. The subtype of the wire delayed signal shall be the same as that of the corresponding port.

The value of a port with an associated wire delay shall be read only in those contexts which are directly related to the modeling of the wire delay itself; that is, the value of the port shall be read only in the context of the actual parameter part of the wire delay concurrent procedure call. The value of the corresponding wire delayed signal is read elsewhere in the model.

A wire delay is applied at the scalar level. Wire delay for a scalar port is modeled with a simple concurrent procedure call.

Wire delay for an array port is modeled with a generate statement of a specific form. The generate statement shall have a generate parameter specification in which the discrete range is a predefined Range attribute, and the prefix of that attribute shall denote the port with the associated wire delay. The only statement within the generate statement shall be a wire delay concurrent procedure call for an element of the port named in the generate parameter specification. The index selecting the element shall be a name denoting the generate parameter.

The actual parameter part of a wire delay concurrent procedure call shall satisfy the following requirements:

— The actual part associated with the input parameter InSig shall be a name denoting a port of mode IN or INOUT.
— The actual part associated with the output parameter OutSig shall be a name denoting an internal signal that satisfies the requirements for a wire delayed signal.
— The actual part associated with the delay value parameter TWire shall be either a locally static value, or a name denoting an interconnect path delay timing generic. The delay value shall be non-negative.

 Copyright © 2001 IEEE. All rights reserved.

If the interconnect path delay generic associated to an instance of a VitalWireDelay call is in the multiple source mode (see 5.2.7.1.5 for details of multiple source mode), the simulation of this call to VitalWireDelay changes as follows:

— On every event on the actual associated to the InSig parameter, select the set of sources of this actual which have changed in this delta cycle.
— For each source load the location (see 5.2.7.1.5 for the details of the location created) created during SDF backannotation. This location can contain 1 (VitalDelayType), 2 (VitalDelayType01) or 6 (VitalDelayType01Z) delays. Apply the delay selection rules outlined in the corresponding VitalCalcDelay function declared in package VITAL_Timing using the new and the old value of the actual of the InSig parameter.

NOTE—This is equivalent to the call <Source Delay> := VitalCalcDelay(<Source Location>, <New Value>, <Old Value>); where the type of <Source Location> has been set depending upon the number of delays in that location.

— Select the minimum of all the source delays.
— The new value of the actual associated to the InSig parameter is assigned to the parameter associated to the OutSig parameter with this selected delay using transport mode.

NOTE—The restrictions on reading the value of a port with an associated wire delay do not preclude the use of the name of the port as a prefix to certain predefined attributes. Use of attributes such as the Range attribute may be necessary to declare an appropriate wire delayed signal or to specify an appropriate range in a generate parameter specification.

6.4.2 Negative constraint block statement

A *negative constraint block* is a special form of a VHDL block statement that is required to model negative timing constraint values (see 8.2 for details on modeling negative timing constraints).

VITAL_negative_constraint_block_statement ::=
  *block*_label :
  block
  begin
    *VITAL*_negative_constraint_block_statement_part
  end block [*block*_label] ;

VITAL_negative_constraint_block_statement_part ::=
  { VITAL_negative_constraint_concurrent_procedure_call
  | VITAL_negative_constraint_generate_statement _part}

VITAL_negative_constraint_generate_statement _part ::=
  VITAL_negative_constraint_generate_statement {
VITAL_negative_constraint_generate_statement}

VITAL_negative_constraint_generate_statement ::=
*generate*_label :
for VITAL_negative_constraint_generate_parameter_specification **generate**
  { VITAL_negative_constraint_concurrent_procedure_call }
end generate {*generate*_label }

VITAL_negative_constraint_generate_parameter_specification ::=
  identifier **in** *range*_attribute_name

VITAL_negative_constraint_concurrent_procedure_call ::= concurrent_procedure_call

The label of a VITAL negative constraint block shall be the name SignalDelay.

Copyright © 2001 IEEE. All rights reserved.

A negative constraint block shall contain exactly one negative constraint concurrent procedure call for each timing generic representing an internal clock delay or an internal signal delay.

A negative constraint concurrent procedure call invokes the procedure VITALSignalDelay that is declared in the package VITAL_Timing. The effect of this call is to delay the associated input port by creating a corresponding internal signal that is delayed by the appropriate amount. A negative constraint block is the only context in which a call to VITALSignalDelay is allowed.

The internal clock delay or internal signal delay for a scalar port is modeled with a VitalSignalDelay procedure call.

Internal clock delay for a vector port shall be modeled with a generate statement of a specific form. The generate statement shall have a generate parameter specification in which the discrete range is a predefined Range attribute, the prefix of the attribute shall denote the clock port associated with the internal clock delay. The only statement within the generate statement shall be a VitalSignalDelay procedure call for an element of the port named in the generate parameter specification. The index selecting the element shall be a name denoting the generate parameter.

Internal signal delay for a vector port shall be modeled with a generate statement of a specific form. The generate statement shall be a either a simple generate statement or a nested generate statement.

The simple generate statement shall have a generate parameter specification in which the discrete range is a predefined Range attribute, the prefix of the attribute shall denote the input port associated with the internal signal delay. The only statement within the generate statement shall be a VitalSignalDelay procedure call for an element of the port named in the generate parameter specification. The index selecting the element shall be a name denoting the generate parameter.

The nested generate statement, shall have a generate parameter specification in which the discrete range is a predefined Range attribute. The prefix of the attribute shall denote either the input port or the clock port. The only statements within the generate parameter specification shall be a simple generate statement.

The formal parameters of the VITALSignalDelay procedure are associated as follows:

- The actual part associated with the delay value parameter Dly shall be a timing generic representing an internal signal delay or an internal clock delay.
- The actual part associated with the input signal parameter InSig shall be a static name denoting either an input port or the corresponding wire delayed signal (if it exists).
- The actual part associated with the output signal parameter OutSig shall be an internal signal. The internal signal shall have the same subtype as the signal associated with the input signal parameter.

6.4.3 VITAL process statement

A VITAL process is a key building block of a Level 1 architecture—it is a mechanism for modeling timing-constraints, functionality, and path delays.

VITAL_process_statement ::=
  [*process*_label :]
  process (sensitivity_list)
    VITAL_process_declarative_part
  begin
    VITAL_process_statement_part
  end process [*process*_label] ;

 Copyright © 2001 IEEE. All rights reserved.

A VITAL process statement shall have a sensitivity list. The sensitivity list shall contain the longest static prefix of every signal name that appears as a primary in a context in which the value of the signal is read. These are the only signal names that the sensitivity list may contain.

6.4.3.1 VITAL process declarative part

A VITAL process declarative part is restricted to a few kinds of declarations.

```
VITAL_process_declarative_part ::=
    { VITAL_process_declarative_item }

VITAL_process_declarative_item ::=
      constant_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | VITAL_variable_declaration
```

6.4.3.1.1 VITAL variables

VITAL variables fall into two classes that are distinguished by the manner in which their members are used. A VITAL *restricted variable* is a variable that is required to store persistent data for private use by certain procedures in the VITAL standard packages. The proper functioning of these procedures requires that the content of the variable not be modified separately by the model itself, hence the use of a restricted variable within a Level 1 architecture is strictly limited. A variable that is not a restricted variable is called an *unrestricted variable*.

```
VITAL_variable_declaration ::=
    variable identifier_list : type_mark [ index_constraint ] [ := expression ] ;
```

The type of an unrestricted variable shall be one of the standard logic types Std_ulogic and Std_logic_vector, or the standard type Boolean.

6.4.3.1.1.1 VITAL restricted variables

A restricted variable is identified by its association with a restricted formal parameter. This single association is the only use permitted for a VITAL restricted variable.

Certain formal parameters of some procedures in the VITAL standard packages are designated as *restricted formal parameters*. They are:

GlitchData — on procedure VitalPathDelay

TimingData — on procedures VitalSetupHoldCheck, VitalRecoveryRemovalCheck

PeriodPulseData — on procedure VitalPeriodPulseCheck

PreviousDataIn — on procedure VitalStateTable

SkewData — on procedures VitalInPhaseSkewCheck, VitalOutPhaseSkewCheck

The actual part in the association of a restricted formal parameter shall be the simple name of a restricted variable.

Copyright © 2001 IEEE. All rights reserved.

Certain restrictions are placed on the declaration of a restricted variable. The type mark in the restricted variable declaration shall denote the type or subtype denoted by the type mark in the corresponding restricted formal parameter declaration. If the declaration of the restricted variable contains an initial value expression then that expression shall take one of the following forms:

— It can be the name of a constant that is declared in one of the VITAL standard packages.
— It can be a function call invoking a function that is declared in one of the VITAL standard packages. Each actual parameter part in such a function call shall be a locally static expression.

NOTE—No initialization is required for GlitchData and PreviousDataIn.

6.4.3.2 VITAL process statement part

The VITAL process statement part consists of statements that describe timing constraint checks, cell function, and path delay selection.

VITAL_process_statement_part ::=
  [VITAL_timing_check_section]
  [VITAL_functionality_section]
  [VITAL_path_delay_section]

These statements are grouped into three distinct sections, each of which is devoted to a particular sort of specification. A VITAL process shall include at least one of these sections.

6.4.3.2.1 Timing check section

The timing check section performs timing constraint checks through the invocation of predefined timing check procedures. Timing checks that can be performed in this section include setup/hold checks, recovery/removal checks, skew checks and period/pulsewidth checks.

VITAL_timing_check_section ::=
  if VITAL_timing_check_condition **then**
    { VITAL_timing_check_statement }
  end if ;

VITAL_timing_check_condition ::= *generic*_simple_name

VITAL_timing_check_statement ::= procedure_call_statement

The timing check condition shall be a simple name denoting a TimingChecksOn control generic that shall be declared in the entity.

A timing check statement is a procedure call statement that invokes one of the VITAL timing check procedures declared in the package VITAL_Timing: VITALSetupHoldCheck, VITALRecoveryRemovalCheck, VitalInPhaseSkewCheck, VitalOutPhaseSkewCheck or VITALPeriodPulseCheck. Each of these procedures performs the specified check and returns a parameter value indicating whether or not a constraint violation occurred (see 8.1 for more detail). These values are considered to be the output of the timing section. A timing check section is the only context in which a call to a timing check procedure is allowed.

The actual parameter part of a timing check procedure call shall satisfy the following requirements:

— The actual part associated with a formal parameter representing a signal name shall be a locally static name.

 Copyright © 2001 IEEE. All rights reserved.

— The actual part associated with the formal parameter HeaderMsg shall be a globally static expression.
— The actual part associated with a formal parameter of the standard type Time shall be a globally static expression.
— The actual part associated with a formal parameter XOn shall be a locally static expression or a simple name denoting the control generics XOn or XOnChecks.
— The actual part associated with a formal parameter MsgOn shall be a locally static expression or a simple name denoting the control generics MsgOn or MsgOnChecks.
— A function call or operator in an actual part shall invoke a function or operator that is defined in package Standard, package Std_Logic_1164, or package VITAL_Timing.
— An actual part associated with the formal parameter TestSignalName, RefSignalName, or RefTransition shall be a locally static expression.

Timing checks shall be independent of one another. It is an error for a variable that is associated with a timing check violation parameter to appear in another timing check statement.

NOTES:

1) Although the variable associated with the violation parameter in a timing check cannot be used in another timing check, it may be used in other sections of the VITAL process — during functionality computation, for instance.

2) It is erroneous for a VITAL Level-1 compliant model to use the control generics XOnGlitch and MsgOnGlitch to associate formal parameters XOn and MsgOn respectively in a call to VITAL timing procedure.

6.4.3.2.2 Functionality section

The functionality section describes the behavior of the cell.

VITAL_functionality_section ::=
  { VITAL_variable_assignment_statement | procedure_call_statement }

VITAL_variable_assignment_statement ::=
  VITAL_target := expression ;

VITAL_target ::= *unrestricted_variable*_name

The function of a model is specified in terms of variable assignment statements and procedure call statements.

A procedure call statement in the functionality section shall invoke the predefined procedure VITALStateTable that is defined in package VITAL_Primitives (see 7.3.4). The actual parameter part of the procedure call shall satisfy the following requirements:

— The actual part associated with the StateTable parameter shall adhere to the restrictions relating to parameter StateTable as described in 6.4.4.
— The constraint on the variable associated with the PreviousDataIn parameter shall match that on the actual associated with the DataIn parameter.

Certain restrictions are placed on a VITAL variable assignment statement. The target shall be an unrestricted variable that is denoted by a locally static name, and the right-hand side expression shall be such that every primary in the right-hand side expression is one of the following:

1) A globally static expression

2) A name denoting a variable, a port, or an internal signal

Copyright © 2001 IEEE. All rights reserved.

3) A function call invoking a standard logic function, a VITAL primitive, or the function VITALTruthTable

4) An aggregate, or a qualified expression whose operand is an aggregate

5) A parenthesized expression

The functionality section is the only context in which a call to the function VITALTruthTable (see 7.3.3) is allowed. The actual part associated with the formal parameter TruthTable in such a call shall adhere to the restrictions relating to parameter TruthTable as described in 6.4.4.

NOTE—The function form of VITALTruthTable can be invoked only from the functionality section; however, the procedure form of VITALTruthTable can be invoked as a VITAL primitive concurrent procedure call.

6.4.3.2.3 Path delay section

The path delay section drives ports or internal signals using appropriate delay values, with provisions for glitch handling, message reporting control, and output strength mapping.

Path delay selection is modeled with a procedure call statement that invokes one of the path delay procedures — VITALPathDelay, VITALPathDelay01, or VITALPathDelay01Z — defined in the package VITAL_Timing. A path delay procedure selects the appropriate propagation delay path and schedules a new output value for the specified signal (see 9.4 for more detail). A path delay section is the only context in which a call to a path delay procedure is allowed.

The actual parameter part of a path delay procedure call shall satisfy the following requirements:

— The actual part associated with the formal parameter OutSignal shall be a locally static name.
— The actual part associated with the formal parameter Paths shall be an aggregate. Each element expression of the array aggregate shall be a record aggregate. The expression associated with a PathDelay subelement shall be globally static. The expression associated with an InputChangeTime subelement shall be either a Last_Event attribute or a locally static expression.
— The actual part associated with the formal parameter GlitchMode shall be a locally static expression.
— The actual part associated with the formal parameter GlitchData shall be a locally static name.
— The actual part associated with a formal parameter XOn shall be a locally static expression or a simple name denoting the control generics XOn or XOnGlitch.
— The actual part associated with a formal parameter MsgOn shall be a locally static expression or a simple name denoting the control generics MsgOn or MsgOnGlitch.
— An actual part associated with the formal parameter OutSignalName, DefaultDelay, or OutputMap shall be a locally static expression.

NOTES:

1) Each port of mode OUT, INOUT, or BUFFER which has a driver is driven by a call to a VITAL primitive procedure or a call to a path delay procedure.

2) It is erroneous for a VITAL Level-1 compliant model to use the control generics XOnChecks and MsgOnChecks to associate formal parameters XOn and MsgOn respectively in a call to VITAL path delay procedure.

6.4.4 VITAL primitive concurrent procedure call

VITAL_primitive_concurrent_procedure_call ::=
*VITAL_primitive*_concurrent_procedure_call

 Copyright © 2001 IEEE. All rights reserved.

A VITAL primitive concurrent procedure call provides a distributed delay modeling capability. It invokes any one of the primitives defined in package VITAL_Primitives to compute functionality and schedule signal values using delay values selected within the procedure. A complete list of the available primitives is given in Clause 7.

The actual parameter part of a primitive subprogram call shall satisfy the following requirements:

— An actual part associated with a formal parameter of class VARIABLE or SIGNAL shall be a static name.
— An actual part associated with a formal parameter of class CONSTANT shall be a globally static expression.
— An actual part associated with the formal parameter ResultMap shall be a locally static expression.
— An actual part associated with a TruthTable or StateTable formal parameter in a call to a table primitive shall be a constant that is not a deferred constant. Furthermore, the value expression of that constant shall be a positional aggregate formed using only locally static expressions or nested aggregates of this form.

Copyright © 2001 IEEE. All rights reserved.

7. Predefined primitives and tables

The VITAL_Primitives standard package defines a number of primitive functions and procedures that provide basic functional support for VITAL Level 1 models.

The set of VITAL primitives consists of logic primitives and utility primitives. Logic primitives perform basic logic operations. Utility primitives support multiple driver resolution and table operations.

7.1 VITAL logic primitives

Each logic primitive is defined in both function and procedure form, for use in the functionality section of a VITAL process or in a VITAL primitive concurrent procedure call, respectively.

The logic primitives consist of the following:

Table 3—VITAL Logic Primitives

VitalAND	VitalAND2	VitalAND3	VitalAND4
VitalOR	VitalOR2	VitalOR3	VitalOR4
VitalXOR	VitalXOR2	VitalXOR3	VitalXOR4
VitalNAND	VitalNAND2	VitalNAND3	VitalNAND4
VitalNOR	VitalNOR2	VitalNOR3	VitalNOR4
VitalXNOR	VitalXNOR2	VitalXNOR3	VitalXNOR4
VitalBUF	VitalBufIf0	VitalBufIf1	VitalIDENT
VitalINV	VitalInvIf0	VitalInvIf1	
VitalMux	VitalMux2	VitalMux3	VitalMux4
VitalDecoder	VitalDecoder2	VitalDecoder4	VitalDecoder8

7.1.1 Logic primitive functions

VITAL logic primitive functions compute the defined function and return a value of type Std_Ulogic or Std_Logic_Vector. All parameters of the logic primitive functions are constants of mode IN.

Example:

```
ARCHITECTURE PinToPinDelay of AndOr IS
    attribute VITAL_LEVEL1 of PinToPinDelay: architecture is TRUE;
BEGIN
    VitalBehavior: PROCESS (A, B, C, D)
        VARIABLE AND1_Out, AND2_Out, Q_Zd: std_ulogic;
        VARIABLE GlitchData_Q : VitalGlitchDataType;
    BEGIN
        -- Functionality section
        AND1_Out  := VitalAND2 ( A, B );
        AND2_Out  := VitalAND2 ( C, D );
        Q_zd    := VitalOR2  (AND1_Out, AND2_Out, ResultMap => DefaultECLMap);
        -- Path delay section
```

 Copyright © 2001 IEEE. All rights reserved.

```
            VitalPathDelay01 (
                OutSignal => Q,
                OutSignalName => "Q",
                OutTemp => Q_zd,
                Paths => (
                    0 => (InputChangeTime => A'last_event,
                          PathDelay => tpd_A_Q,
                          PathCondition => TRUE),
                    1 => (InputChangeTime => B'last_event,
                          PathDelay => tpd_B_Q,
                          PathCondition => TRUE),
                    2 => (InputChangeTime => C'last_event,
                          PathDelay => tpd_C_Q,
                          PathCondition => TRUE),
                    3 => (InputChangeTime => D'last_event,
                          PathDelay => tpd_D_Q,
                          PathCondition => TRUE)
               ),
                GlitchData => GlitchData_Q,
                DefaultDelay => VitalZeroDelay01,
                Mode => OnDetect,
                XoN => TRUE,
                MsgOn => TRUE,
                MsgSeverity => WARNING);
        END PROCESS;
    END;
```

7.1.2 Logic primitive procedures

VITAL logic primitive procedures execute in a manner similar to that of a separate process. The procedures wait internally for an event on an input signal, compute the new result, perform glitch handling, schedule transactions on the output signals, and wait for further input events. All of the functional (logic) input or output parameters of the primitive procedures are signals. All other parameters are constants.

The procedure primitives are parameterized for separate path delays from each input signal. All path delays default to 0 ns.

Example:

```
    ARCHITECTURE DistributedDelay OF AndOr IS
        ATTRIBUTE VITAL_LEVEL1 of DistributedDelay: architecture IS TRUE;
        SIGNAL AND1_Out, AND2_Out: std_ulogic;
    BEGIN
        I1: VitalAND2 (AND1_Out, A, B, tdevice_I1_Q, tdevice_I1_Q);
        I2: VitalAND2 (AND2_Out, C, D, tdevice_I2_Q, tdevice_I2_Q);
        I3: VitalOR2   (Q, AND1_Out, AND2_Out, tdevice_I3_Q, tdevice_I3_Q);
    END;
```

7.1.3 Establishing output strengths

Each logic primitive function or procedure by default produces an output value from the set of values { ‘U’, ‘X’, ‘0’, ‘1’, ‘Z’}. This set of logic strengths may be expanded through use of the optional ResultMap parameter (of type ResultMapType), which provides rapid conversion of any of the standard five output values to any other output value through the use of a simple table lookup.

Copyright © 2001 IEEE. All rights reserved.

```
type ResultMapType is array (UX01) of Std_ulogic;
type ResultZMapType is array (UX01Z) of Std_ulogic;

constant VitalDefaultResultMap  : VitalResultMapType := ( 'U', 'X', '0', '1' );
constant VitalDefaultResultZMap : VitalResultZMapType  := ( 'U', 'X', '0', '1', 'Z' );
```

Example:

```
ARCHITECTURE Structural OF Pullup IS
    ATTRIBUTE VITAL_LEVEL1 of Structural: architecture IS TRUE;
    CONSTANT DefaultECLMap : VitalResultMapType := (‘U’, ‘X’, ‘L’, ‘1’);
BEGIN
    I1: VitalBUF (Q, A, tpd_A_Q, ResultMap => DefaultECLMap);
END;
```

In this example, the constant DefaultECLMap that is supplied as the ResultMap actual parameter causes a ‘0’ is to be mapped to ‘L’ on the output of the primitive.

7.2 VitalResolve

The procedure VitalResolve supports the resolution of multiple signal drivers, allowing a model to drive them on a single signal. It invokes the standard logic function Resolved on the input vector and assigns it to the outputs with a zero delay.

Example:

```
ARCHITECTURE Structural OF ResolvedLogic IS
    ATTRIBUTE VITAL_LEVEL1 of Structural: architecture IS TRUE;
    SIGNAL Q_Temp1, Q_Temp2 : std_ulogic;
BEGIN
    I1: VitalAND2 (Q_Temp1, A, B, tdevice_I1_Q, tdevice_I1_Q);
    I2: VitalAND2 (Q_Temp2, C, D, tdevice_I2_Q, tdevice_I2_Q);
    R1: VitalResolve (Q, (Q_Temp1, Q_Temp2));
END;
```

7.3 VITAL table primitives

Package VITAL_Primitives supports the standard specification and use of truth tables and symbol tables through the use of the table primitives VitalTruthTable and VitalStateTable. VitalTruthTable is provided for modeling combinational cells. VitalStateTable is provided for modeling sequential cells.

7.3.1 VITAL table symbols

A transition set or a steady state condition is represented by a special table symbol. The symbol set defined by the type VitalTableSymbolType is used to specify high accuracy state tables.

```
type VitalTableSymbolType is (
    '/',    -- 0 -> 1
    '\',    -- 1 -> 0
    'P',    -- Union of '/' and '^' (any edge to 1)
    'N',    -- Union of '\' and 'v' (any edge to 0)
    'r',    -- 0 -> X
    'f',    -- 1 -> X
```

 Copyright © 2001 IEEE. All rights reserved.

```
    'p',    -- Union of '/' and 'r' (any edge from 0)
    'n',    -- Union of '\' and 'f' (any edge from 1)
    'R',    -- Union of '^' and 'p' (any possible rising edge)
    'F',    -- Union of 'v' and 'n' (any possible falling edge)
    '^',    -- X -> 1
    'v',    -- X -> 0
    'E',    -- Union of 'v' and '^' (any edge from X)
    'A',    -- Union of 'r' and '^' (rising edge to or from 'X')
    'D',    -- Union of 'f' and 'v' (falling edge to or from 'X')
    '*',    -- Union of 'R' and 'F' (any edge)
    'X',    -- Unknown level
    '0',    -- low level
    '1',    -- high level
    '-',    -- don't care
    'B',    -- 0 or 1
    'Z',    -- High Impedance
    'S'     -- steady value
  );
```

The acceptable range of table symbols for each sort of table is defined with a scalar subtype definition. A truth or state table can be constructed from any value in the subset of values defined by the corresponding table symbol subtype.

```
subtype VitalTruthSymbolType is VitalTableSymbolType range 'X' to 'Z';
subtype VitalStateSymbolType is VitalTableSymbolType range '/' to 'S';
```

Table 4 shows the VitalTableSymbolType elements and the levels and edge transitions that they represent.

Table 4—Truth and State Table symbol semantics

	00	10	X0	11	01	X1	0X	1X	XX
‘/‘					*				
‘\‘		*							
‘P’					*	*			
‘N’		*	*						
‘r’							*		
‘f’								*	
‘p’					*		*		
‘n’		*						*	
‘R’					*	*	*		
‘F’		*	*					*	
‘^’						*			
‘v’			*						
‘E’			*			*			
‘A’						*	*		
‘D’			*					*	
‘*’		*	*		*	*	*	*	
‘X’							*	*	*
‘0’	*	*	*						
‘1’				*	*	*			
‘-’	*	*	*	*	*	*	*	*	*
‘B’	*	*	*	*	*	*			
‘Z’									
‘S’	*			*					

Copyright © 2001 IEEE. All rights reserved.

A truth or state table is partitioned into different sections, each of which represents a specific kind of information. These sections include an *input pattern* and a *response*. For a state table, an additional *state* section is included. The input pattern section shall not contain the symbol ‘Z’. The response section shall contain only the symbols ‘X’, ‘0’, ‘1’, ‘-’ and ‘Z’, and for a state table, the symbol ‘S as well. The state section of a state table can only contain the symbols ‘X’, ‘0’, ‘1’, ‘-’ and ‘B’. It is an error if any symbols other than the allowed ones are encountered in a section.

NOTES:

1) The table symbols are enumeration literals, therefore they are case-sensitive.

2) A limited set of table symbol values can be used to develop truth tables. Any table symbol can be used in a state table.

7.3.2 Table symbol matching

During truth or state table processing, the input to the table primitive (DataIn) is matched to the stimulus portion of the table. The matching process begins by converting the input data to the equivalent ‘X’, ’0’, or ’1’ values by applying the standard logic TO_X01 function. The resulting values are then compared to the stimulus portion of the table according to the following matching rules:

Table 5—Matching of Table Symbols to Input Stimulus

Table Stimulus Portion	DataInX01 := To_X01(DataIn)	Result of comparison
‘X’	‘X’	‘X’ only matches with ‘X’
‘0’	‘0’	‘0’ only matches with ‘0’
‘1’	‘1’	‘1’ only matches with ‘1’
‘-’	‘X’, ‘0’, ‘1’	‘-’ matches with any value of DataInX01
‘B’	’0’, ‘1’	‘B’ only matches with ‘0’ or ‘1’

For a state table, the current and previous values of DataIn are used to determine if an edge has occurred. These edges are matched with the edge entries which are specified in the input pattern of the table using the semantics of the edge symbols shown in Table 4.

7.3.3 TruthTable primitive

A function version of VitalTruthTable is defined for use inside a VITAL process. The procedure version of VitalTruthTable is defined for use in a concurrent procedure call. In addition to performing the same result computation as the function version, the procedure version schedules the resulting value on the output signal with a delay of 0 ns. Overloaded forms are provided to support both scalar and vector output.

7.3.3.1 Truth table construction

A VITAL truth table is an object of type VitalTruthTableType.

```
  type VitalTruthTableType is array ( Natural range <>, Natural range <>) of
VitalTruthSymbolType;
```

 Copyright © 2001 IEEE. All rights reserved.

The length of the first dimension of a truth table is the number of input combinations that have a specified output value. The length of the second dimension shall be the sum of the size of the input pattern section and the size of response section.

The number of inputs to the truth table shall be equal to the length of the DataIn parameter. It is an error if the length of DataIn is greater than or equal to the size of the second dimension of the TruthTable parameter.

A row in a truth table consists of two sections: an *input pattern* and a *response*. A row *i* of the truth table is interpreted as follows:

InputPattern(j downto 0), Response(k downto 0)
where,
j = DataIn'Length - 1
k = TruthTable'Length(2) - DataIn'Length - 1

Example:

Truth table for a 2 to 4 decoder:

```
Constant DecoderTable: VitalTruthTableType(0 to 3, 0 to 5) :=
--   Input Pattern       Response
--   D1      D0          Q3  Q2  Q1  Q0

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

7.3.3.2 TruthTable algorithm

The VitalTruthTable primitive compares the stimulus, DataIn, with the input pattern section of each row (starting from the top) in TruthTable, to find the first matching entry. If all of the subelements of DataIn match with corresponding subelements of the input pattern of a particular row in TruthTable, the outputs are determined from the response section of the corresponding row. The outputs are then converted to the standard logic X01Z subtype. If all rows in TruthTable are searched and no match is found, then VitalTruthTable returns either an 'X' or a vector of 'X's, as appropriate.

The vector form of the procedure places the outputs in the actual associated with the parameter Result, starting from the right side of both the truth table and the actual associated with Result, until the actual is filled or there are no more outputs left in the truth table. It is an error if Result is too small or too large to hold all of the values. The vector function behaves in a manner similar to the vector procedure; however, it always returns a vector with the range TruthTable'Length(2) - DataIn'Length - 1 downto 0.

7.3.4 StateTable primitive

There are two versions of the VitalStateTable procedure — one which is intended for use as a sequential statement and one which is intended for use as a concurrent statement. The concurrent statement version of this procedure performs the same result computation as the function version, but in addition it schedules the resulting value on the output signal with a delay of 0 ns. Overloaded forms are provided to support both scalar and vector output.

Copyright © 2001 IEEE. All rights reserved.

7.3.4.1 State table construction

A VITAL state table is an object of type VitalStateTableType.

> **type** VitalStateTableType **is array** (Natural **range** <>, Natural **range** <>) **of** VitalStateSymbolType;

The length of the first dimension of a state table is the number of input and state combinations that have a specified output value. The length of the second dimension is the sum of the length of the input patterns section, the length of the state section, and the length of the response section.

The number of inputs to the state table shall equal the length of the DataIn parameter. It is an error if the length of DataIn is greater than or equal to the size of the second dimension of the StateTable parameter.

A row in a state table consists of the following sections: an *input pattern*, a *state* and a *response*. Each row in the table shall have at most one element from the subtype VitalEdgeSymbolType. Each row *i* of the StateTable is interpreted as follows:

InputPattern(*j* downto 0), State(*k* downto 0), Response(*l* downto 0)
where,
j = DataIn'Length - 1
k = NumState - 1
l = StateTable'Length(2) - DataIn'Length - NumState -1

NOTE—A state table should include at least one entry with an 'S' for the clock so that VitalStateTable can handle the case in which the procedure is activated but the clock did not change. If this entry is not included, then the result defaults to 'X's.

Example:

State table for a positive-edge triggered D Flip flop:

```
Constant DFFTable: VitalStateTableType :=
--   RESET    D     CLK   State   Q

((   '0',     '-',  '-',  '-',    '0'),
(    '1',     '1',  '/',  '-',    '1'),
(    '1',     '0',  '/',  '-',    '0'),
(    '1',     'X',  '/',  '-',    'X'),
(    '1',     '-',  '-',  '-',    'S'));
```

7.3.4.2 StateTable algorithm

The procedure VitalStateTable computes the value of the output of a synchronous sequential circuit (a Moore machine) based on the inputs, the present state, and a state table. These procedures compare the stimulus, DataIn (and edges on it) with the input pattern section of each row (starting from the top) in StateTable, to find the first matching entry. If all input entries are found to match, the comparison moves to the states. Here the comparison moves from the leftmost index of Result (comparing it to State(NumStates - 1) in the state table) and proceeds to the right. The comparison of the entry continues until all of the inputs have been compared or a mismatch is encountered. The search terminates with the first level or edge match or when the table entries are exhausted. If all rows in StateTable are searched and no match is found, then the actual associated with the formal parameter Result is assigned an 'X' or a vector of 'X's, as appropriate.

 Copyright © 2001 IEEE. All rights reserved.

Once a match is found, or it is determined that no match can be made, the new values of the state variables and the outputs are determined from the response section of the state table. The states and outputs are placed into the parameter Result, starting from the right side of both the state table and Result, until Result is filled or there are no more outputs or states left in the state table. It is an error if Result is too small or too large to hold all of the values.

Copyright © 2001 IEEE. All rights reserved.

8. Timing constraints

The VITAL ASIC modeling specification provides support for standard timing constraint checking and for the modeling of negative timing constraints.

8.1 Timing check procedures

Package VITAL_Timing defines five kinds of timing check procedures: VitalSetupHoldCheck, VITALRecoveryRemovalCheck, VitalInPhaseSkewCheck, VitalOutPhaseSkewCheck and VITALPeriodPulseCheck. VitalSetupHoldCheck is overloaded for use with test signals of type Std_Ulogic or Std_Logic_Vector. Each defines a CheckEnabled parameter that supports the modeling of conditional timing checks.

A VITAL timing check procedure performs the following functions:

— Detects a timing constraint violation if the timing check is enabled.
— Reports a timing constraint violation using a VHDL assertion statement. The report message and severity level of the assertion are controlled by the model.
— Sets the value of a corresponding violation flag. If a timing violation is detected, the value of this flag is set to ‘X’, otherwise it is set to ‘0’. ‘X’ generation for this flag can be controlled by the model.

The same timing check procedures are used for both positive and negative timing constraint values. Two delay parameters —TestDelay and RefDelay — are defined for modeling the delays associated with the test or reference signals when negative setup or hold constraints are in effect. The delay parameters shall have the value zero when negative constraints do not apply.

8.1.1 VitalSetupHoldCheck

The procedure VitalSetupHoldCheck detects the presence of a setup or hold violation on the input test signal with respect to the corresponding input reference signal. The timing constraints are specified through parameters representing the high and low values for the setup and hold times. This procedure assumes non-negative values for setup/hold timing constraints.

Setup constraint checks are performed by this procedure only if the CheckEnabled condition evaluates to True at the reference edge. Hold constraint checks are performed by this procedure only if the CheckEnabled condition evaluates to True at the test edge. The event times required for constraint checking are always updated, regardless of the value of CheckEnabled. Four optional parameters are provided for explicit control over condition checking at the test and reference edges. A setup check will be performed only if EnableSetupOnTest is True at the test edge and EnableSetupOnRef is True at the reference edge. A hold check will be performed only if EnableHoldOnRef is True at the reference edge and EnableHoldOnTest is True at the test edge. Setup constraints are checked in the simulation cycle in which the reference edge occurs. A setup violation is detected if the time since the last TestSignal change is less than the expected setup constraint time. Hold constraints are checked in the simulation cycle in which an event on TestSignal occurs. A hold violation is detected if the time since the last reference edge is less than the expected hold constraint time.

 Copyright © 2001 IEEE. All rights reserved.

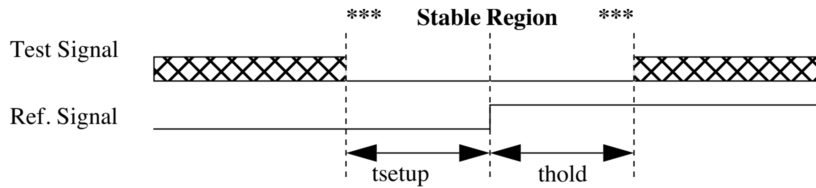


Figure 2—Setup/hold check for positive constraints

8.1.2 VitalPeriodPulseCheck

The procedure VitalPeriodPulseCheck checks for minimum periodicity and pulse width for '1' and '0' values of the input test signal. The timing constraint is specified through parameters representing the minimal period between successive rising or falling edges of the input test signal, and the minimum pulse widths associated with high and low values.

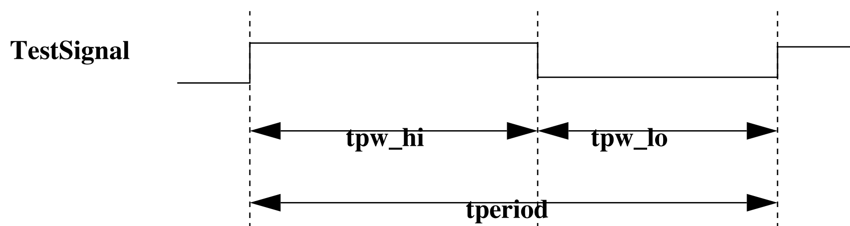


Figure 3—Period/Pulsewidth check

8.1.3 VitalRecoveryRemovalCheck

The procedure VitalRecoveryRemovalCheck detects the presence of a recovery or removal violation on the input test signal with respect to the corresponding input reference signal. It assumes non-negative values of recovery/removal timing constraints. The timing constraint is specified through parameters representing the recovery and removal times associated with a reference edge of the reference signal. A flag indicates whether a test signal is asserted when it is high or when it is low.

Recovery constraint checks are performed by this procedure only if the CheckEnabled condition evaluates to True at the reference edge. Removal constraint checks are performed by this procedure only if the CheckEnabled condition evaluates to True at the test edge. The event times required for constraint checking are always updated, regardless of the value of CheckEnabled. Four optional parameters are provided for explicit control over condition checking at the test and reference edges. A recovery check will be performed only if EnableRecOnTest is True at the test edge and EnableRecOnRef is True at the reference edge. A removal check will be performed only if EnableRemOnRef is True at the reference edge and EnableRemOnTest is True at the test edge. Recovery constraints are checked in the simulation cycle in which the reference edge occurs. A recovery violation is detected if the time since the last TestSignal change is less than the expected recovery constraint time. Removal constraints are checked in the simulation cycle in which an event on TestSignal occurs. A removal violation is detected if the time since the last reference edge is less than the expected removal constraint time.

Copyright © 2001 IEEE. All rights reserved.

8.1.4 Boundary condition behavior

The VITAL timing check procedures define specific behavior of signal changes at the constraint boundary which are as follows:

- No violations are reported if the signals change at the constraint boundary.
- The behavior of the timing constraint checks is consistent irrespective of any delta cycle difference between the signals.
- In case of positive setup/recovery constraint values and zero or negative hold/removal constraint values, only setup/recovery violations will be reported.
- In case of positive hold/removal constraint values and zero or negative setup/recovery constraint values, only hold/removal violations will be reported.
- In case of positive setup/recovery and positive hold/removal, both types of violations will be reported. If the test port and reference port change simultaneously, a hold/removal violation will be reported.

8.1.5 VITAL skew checks

The VITAL skew timing checks detect the presence of skew violations between two input signals Signal1 and Signal2 in any direction. They assume non-negative values of skew timing constraints. Timing constraints are specified through parameters representing skew relations between any edges of the input signals.

The VITAL skew procedures use a Trigger signal of type std_ulogic, which shall be present in the sensitivity list of the process in which the skew timing check occurs. The Trigger signal shall not be used in any other context. A skew violation is detected if the time since the last change in Signal1(Signal2) is greater than the specified skew constraint time, resulting in an event on the Trigger signal. Skew timing constraint checks are performed only if the CheckEnabled condition evaluates to True.

8.1.5.1 VitalInPhaseSkewCheck

The procedure VitalInPhaseSkewCheck detects a skew violation if Signal1 and Signal2 are in different states for an interval greater than the specified skew timing constraint. Four skew timing constraint parameters can be specified in this procedure.

SkewS1S2RiseRise is the absolute maximum time for which Signal2 is allowed to remain in the low state after Signal1 goes to the high state.

SkewS2S1RiseRise is the absolute maximum time for which Signal1 is allowed to remain in the low state after Signal2 goes to the high state.

SkewS1S2FallFall is the absolute maximum time for which Signal2 is allowed to remain in the high state after Signal1 goes to the low state.

SkewS2S1FallFall is the absolute maximum time for which Signal1 is allowed to remain in the high state after Signal2 goes to the low state.

 Copyright © 2001 IEEE. All rights reserved.

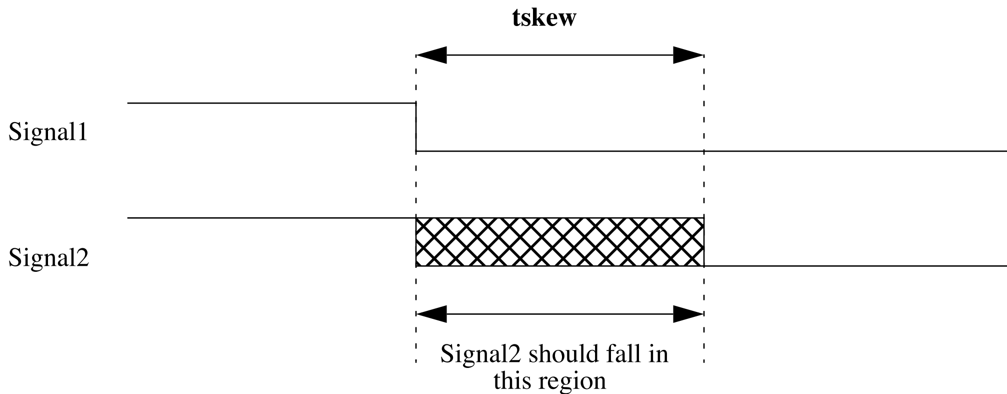


Figure 4—In-phase skew check

8.1.5.2 VitalOutPhaseSkewCheck

The procedure VitalOutPhaseSkewCheck detects a skew violation if Signal and Signal2 are in the same state for an interval greater than the specified skew timing constraint. Four skew timing constraint parameters can be specified in this procedure.

SkewS1S2RiseFall is the absolute maximum time for which Signal2 is allowed to remain in the high state after Signal1 goes to the high state.

SkewS2S1RiseFall is the absolute maximum time for which Signal1 is allowed to remain in the high state after Signal2 goes to the high state.

SkewS1S2FallRise is the absolute maximum time for which Signal2 is allowed to remain in the low state after Signal1 goes to the low state.

SkewS2S1FallRise is the absolute maximum time for which Signal1 is allowed to remain in the low state after Signal2 goes to the low state.

Copyright © 2001 IEEE. All rights reserved.

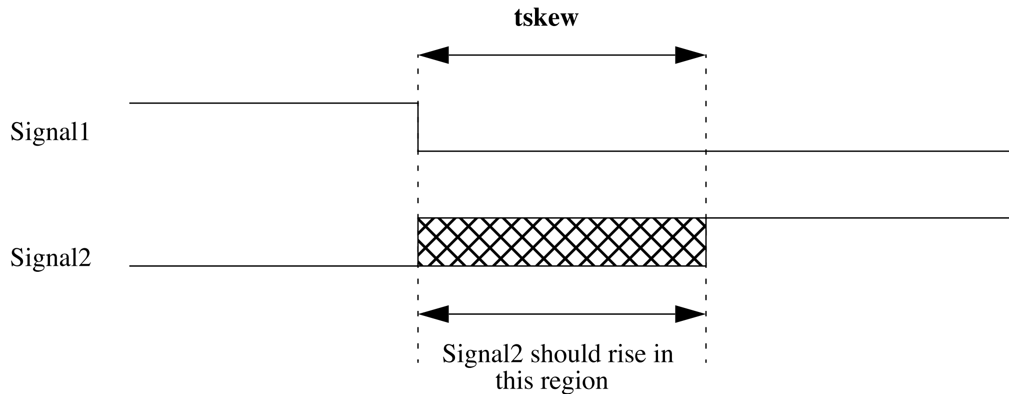


Figure 5—Out-of-phase skew check

NOTE—If SignalDelay block signals are used, there may be situations where the original phase relations between the input signals are no longer valid for the delayed signals, due to internal delays introduced during the negative constraint calculation phase (see 8.2.2). In such situations these procedures may not detect certain skew violations. The non-delayed signals may be used in such situations.

Example:

```
-- process sensitivity list including the input signals and trigger signals for skew checks
VITALBehavior: PROCESS (D, CLK, CLK1, CLK2, RESET, TRG1, TRG2)
    VARIABLE SetupHoldInfo : VitalTimingDataType := VitalTimingDataInit ; --Restricted
variable
    VARIABLE PeriodDataInfo : VitalPeriodDataType := VitalPeriodDataInit ; --Restricted
variable
    VARIABLE RecoRemoInfo : VitalTimingDataType := VitalTimingDataInit ; --Restricted
variable
    VARIABLE SkewInfo1 : VitalSkewDataType := VitalSkewDataInit;    --          Restricted
variable
    VARIABLE SkewInfo2 : VitalSkewDataType := VitalSkewDataInit;    --          Restricted
variable
    VARIABLE Violation_flag, Viol1, Viol2, Viol3, Viol4, Viol5 : X01;
    ...
BEGIN
    -- Timing Check Section
    IF (TimingChecksOn) THEN
        -- Setup/hold check between D and rising CLK
        VitalSetupHoldCheck (
            TestSignal      =>  D,              TestSignalName    =>  ”D”,
            RefSignal       =>  CLK,            RefSignalName     =>  ”CLK”,
            SetupHigh       =>  tsetup_D_CLK,   SetupLow          =>  tsetup_D_CLK,
            HoldHigh        =>  thold_D_CLK,    HoldLow           =>  thold_D_CLK,
            CheckEnabled    =>  RESET = ‘1’,    RefTransition     =>  ‘R’,
            MsgOn           =>  TRUE,           XOn               =>  TRUE,
            HeaderMsg       =>  “Instance1”,    TimingData        =>  SetupHoldInfo,
            Violation       =>  Viol1,          MsgSeverity       =>  ERROR);
        -- Pulsewidth and period check for CLK
```

 Copyright © 2001 IEEE. All rights reserved.

```
        VitalPeriodPulseCheck (
            TestSignal        =>  CLK,            TestSignalName      =>  ”CLK”,
            Period            =>  tperiod_CLK,
            PulseWidthHigh=>  tpw_CLK_posedge,
            PulseWidthLow =>  tpw_CLK_negedge,
            PeriodData        =>  PeriodDataInfo, Violation           =>  Viol2,
            MsgOn             =>  TRUE,           XOn                 =>  TRUE,
            HeaderMsg         =>  “Instance1”,
            CheckEnabled  =>  RESET = ‘1’,    MsgSeverity         =>  ERROR);
        -- Recovery/removal check between RESET and rising CLK
        VitalRecoveryRemovalCheck (
            TestSignal        =>  RESET,          TestSignalName      =>  ”RESET”,
            RefSignal         =>  CLK,            RefSignalName       =>  ”CLK”,
            Recovery          =>  trecovery_RESET_CLK,
            Removal           =>  tremoval_RESET_CLK,
            ActiveLow         =>  FALSE,          CheckEnabled        =>  RESET = ‘1’,
            RefTransition     =>   ‘R’,
            MsgOn             =>  TRUE,           XOn                 =>  TRUE,
            HeaderMsg         =>  “Instance1”,    TimingData          =>  RecoRemoInfo,
            Violation         =>  Viol3,          MsgSeverity         =>  ERROR);
        -- In-phase skew check on CLK1 with respect to CLK
        VitalInPhaseSkewCheck (
            Signal1           =>  CLK,            Signal1Name         =>  “CLK”,
            Signal2           =>  CLK1,           Signal2Name         =>  “CLK1”,
            SkewS1S2RiseRise=>  tskew_CLK_CLK1,
            SkewS1S2FallFall=>    tskew_CLK_CLK1,
            SkewData          =>  SkewInfo1,
            CheckEnabled  =>  TRUE,
            Violation         =>  Viol4,
            Trigger           =>  TRG1);
        -- Out-of-phase skew check between CLK and CLK2
        VitalOutPhaseSkewCheck (
            Signal1           =>  CLK,            Signal1Name         =>  “CLK”,
            Signal2           =>  CLK2,           Signal2Name         =>  “CLK2”,
            SkewS1S2RiseFall  =>  tskew_CLK_CLK2_posedge_negedge,
            SkewS1S2FallRise=>    tskew_CLK_CLK2_negedge_posedge,
            SkewS2S1RiseFall  =>  tskew_CLK2_CLK_posedge_negedge,
            SkewS2S1FallRise  =>  tskew_CLK2_CLK_negedge_posedge,
            SkewData          =>  SkewInfo2,
            Violation         =>  Viol5,
            Trigger           =>  TRG2);
    END IF;
    Violation_flag := Viol1 or Viol2 or Viol4 or Viol5;
    ...
END PROCESS;
```

8.2 Modeling negative timing constraints

Some devices may be characterized with negative setup or hold times, or negative recovery or removal times. If any of these values is negative, then the data constraint interval does not overlap the reference clock edge, and a *negative timing constraint* is said to be *in effect*.

Copyright © 2001 IEEE. All rights reserved.

A negative hold or removal time corresponds to an internal delay on the test (or data) signal. A negative setup or recovery time corresponds to an internal delay on the reference (or clock) signal. These internal delays determine when a data signal is sampled on the edge of the clock signal. Special adjustments are required in the case of negative timing constraints because the data value at the time that the clock edge is detected may be different from the data value during the constraint interval. Furthermore, the setup time may be difficult to check because a violating data edge may not be the most recent data edge preceding the clock.

Negative timing constraints in a VITAL Level 1 and VITAL Level 1 Memory model are handled by internally delaying the test or reference signals. Negative setup or recovery times result in a delayed reference signal. Negative hold or removal times result in a delayed test signal. Furthermore, the delays associated with other signals may need to be appropriately adjusted so that all constraint intervals overlap the delayed reference signals. After these delay adjustments are performed, the timing constraint values on the timing check procedures are always non-negative.

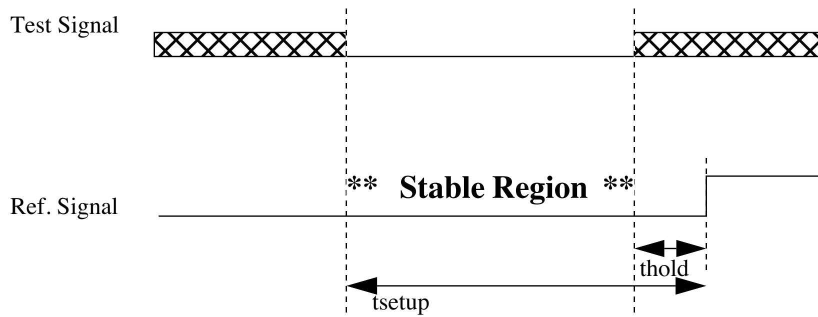


Figure 6—The data constraint interval for a negative hold constraint

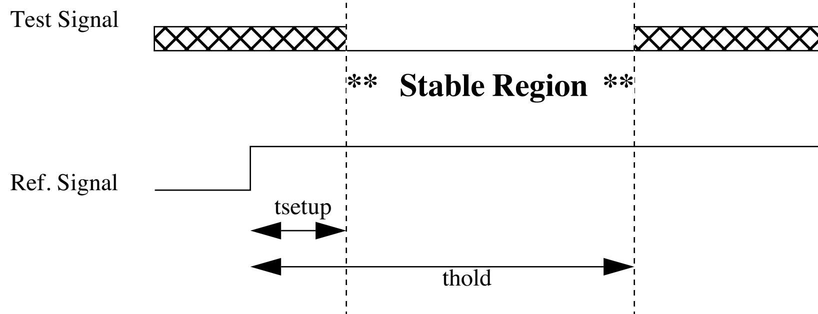


Figure 7—The data constraint interval for a negative setup constraint

8.2.1 Requirements on the VHDL description

The VITAL ASIC modeling specification defines a delay adjustment algorithm that transforms negative delay values to non-negative ones. This algorithm relies on certain model characteristics in order to correctly calculate the delays, therefore a negative timing constraint must be anticipated and the model structured to accommodate them.

 Copyright © 2001 IEEE. All rights reserved.

To model negative constraints in a VITAL compliant model, the corresponding VHDL description shall contain the following:

— The declaration of an internal clock delay generic for each clock (reference) signal that is associated with a negative setup (or recovery) constraint
— The declaration of an internal signal delay generic for every data (test) signal that is associated with a negative hold (or removal) constraint
— The declaration of biased propagation delay generics for paths which are dependent on multiple clocks
— A signal delay block in the architecture which contains calls to the VitalSignalDelay procedure to delay the appropriate test/reference signal

These rules are part of the Level 0 modeling specification (see Clause 4).

NOTE—In general, a model should read the value of the internally delayed signal rather than that of the corresponding signal which is not delayed; however, the model is not prohibited from reading the value of the signal which is not delayed.

8.2.2 Negative constraint calculation phase

The negative constraint delay adjustments are computed outside of the actual VHDL model in a special phase of simulation called the *negative constraint calculation phase*, which occurs directly after the VITAL back annotation phase and directly before normal VHDL initialization.

Negative constraint calculation is performed for each VITAL Level 0 instance which defines a negative constraint timing generic. The values of certain timing generics are computed and set, and the values of others are adjusted in an iterative algorithm that uses the generic values set during previous steps.

Negative constraint calculation is performed in the following sequence:

1) Calculate internal clock delays

2) Calculate internal signal delays

3) Calculate biased propagation delays

4) Adjust propagation delays

5) Adjust timing constraint values corresponding to setup, hold, recovery, and removal times

It is an error if at the end of the negative constraint calculation stage, a timing generic that is adjusted by this algorithm still has a negative value.

For use in the negative constraint calculation of vector timing generics, the following terms are defined in table 6.

Copyright © 2001 IEEE. All rights reserved.

Table 6—Negative Constraint Calculation Definitions

Term	Description
TicdGenericIndex	the index of an internal clock delay generic subelement
TisdGenericIndex	the index of an internal signal delay generic subelement
TbpdGenericIndex	the index of a biased propagation delay generic subelement
TsetupRecoveryGenericIndex	the index of a setup or recovery generic subelement
TholdRemovalGenericIndex	the index of a hold or removal generic subelement
TpdGenericIndex	the index of a propagation delay generic subelement
TestPortIndex	the index of a test port subelement
RefPortIndex	the index of a reference port subelement
RefPortSize	the size of the reference port
OutputPortSize	the size of an output port named in a propagation delay generic.
TpdGenericSize	the size of the propagation delay generic
CrossArc	a vector timing arc whose size is equal to the product of the sizes of the two ports corresponding to the timing generic
ParallelArc	a vector timing arc whose size is equal to the individual sizes of the two ports corresponding to the timing generic

NOTES

1—A calculation or adjustment that is performed as a part of the negative constraint calculation phase may result in a reduction in the value of a generic (or one of its subelements) that causes the value to become negative, in which case the negative constraint algorithm replaces the negative value with a zero value. This situation may or may not indicate an error, hence a tool which processes VITAL compliant models may choose to issue a warning when it replaces the negative value.

2— A vector propagation delay generic or a vector timing check arc should be either a ParallelArc or a CrossArc. It is an error if the size of the generic is different from what is expected in a ParallelArc or a CrossArc (see Table 6).

8.2.2.1 Calculation of internal clock delays

The value of each internal clock delay generic is computed as follows:

1) The name of the associated clock signal is extracted from the <ClockPort> portion of the internal clock delay generic name.

If the internal clock delay generic is a scalar:

2) All setup and recovery timing generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked.

 Copyright © 2001 IEEE. All rights reserved.

3) The minimum value of all the subelements of all the marked timing generics is determined. If that value is negative, the internal clock delay generic receives the absolute value, otherwise it is set to 0 ns.

If the internal clock delay generic is a vector:

For each subelement of the internal clock delay generic:

4) All setup and recovery timing generics on the same instance are examined. Mark those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name.

   For each such setup/recovery generic, the index of the subelements to be marked is calculated as follows:

   If the generic is a CrossArc,

   $$Index = TicdGenericIndex + n * RefPortSize$$

   where $n = 0, 1, 2, \ldots (RefPortSize - 1)$

   If the generic is a ParallelArc,

   $$Index = TicdGenericIndex$$

5) The minimum value of all the marked subelements of all the marked timing generics is determined. If that value is negative, the internal clock delay generic receives the absolute value, otherwise it is set to 0 ns.

8.2.2.2 Calculation of internal signal delays

The value of each internal signal delay generic is computed as follows:

1) The names of the associated clock and input signals are extracted from the <ClockPort> and <InputPort> portions of the internal signal delay generic name.

If the internal signal delay generic is a scalar:

2) If there is an internal clock delay generic containing the same clock signal name, its value is the associated clock delay. Otherwise, the associated clock delay is 0 ns.

3) All hold and removal timing generics on the same instance are examined. Those generics for which the <ReferencePort> part of the name is the same as the <ClockPort> name and the <TestPort> part is the same as the <InputPort> name are marked.

4) The minimum value of all subelements of all the marked timing generics is determined. This value is reduced by the associated clock delay. If the resulting value is negative, it is replaced by its absolute value; otherwise, it is replaced by 0 ns.

If the internal signal delay generic is a vector:

For each subelement of the internal signal delay generic:

5) If there is an internal clock delay generic containing the same clock signal name and it is a scalar, its value is taken as the associated clock delay.

Copyright © 2001 IEEE. All rights reserved.

If there is an internal clock delay generic containing the same clock signal name, and it is a vector the index of the subelement is calculated as follows:

If the internal signal delay generic is a CrossArc,

Index = (TisdGenericIndex) modulus *(RefPortSize)*.

If the internal signal delay generic is a ParallelArc,

Index = TisdGenericIndex

The value of the subelement whose index is computed above is taken as the associated clock delay.

If there is no internal clock delay generic containing the same clock signal name, the associated clock delay is set to 0 ns.

6) All hold and removal timing generics on the same instance are examined. Mark those generics for which the <ReferencePort> part of the name is the same as the <ClockPort> name and the <TestPort> part is the same as the <InputPort> name.

   For each such hold/removal generic, mark the subelement whose index is calculated as follows:

   Index = TisdGenericIndex

7) The minimum value of all marked subelements of all the marked timing generics is determined. This value is reduced by the associated clock delay. If the resulting value is negative, it is replaced by its absolute value; otherwise, it is replaced by 0 ns.

8.2.2.3 Calculation of biased propagation delays

The value of each biased propagation delay generic is computed as follows:

If the biased propagation delay generic is a scalar:

1) The corresponding propagation delay generic (denoting the same input and output ports, condition name, and edge) is identified (see 4.3.2.1.3.14). The value of the biased propagation delay generic is initialized to the value of the corresponding propagation delay generic.

2) The names of the associated clock and input signals are extracted from the <ClockPort> and <InputPort> portions of the biased propagation delay generic name.

3) If there is an internal signal delay generic (see 4.3.2.1.3.13) on the same instance whose name denotes the same <InputPort> and <ClockPort> parts, then the value of each subelement of the biased propagation delay generic is reduced by the value of that internal signal delay generic. If the resulting value of any subelement is negative then the value of that subelement is set to zero.

NOTE—Due to the name construction of the internal signal delay generic, there can be only one internal signal delay generic that matches both the InputPort and ClockPort names (in step 3).

If the biased propagation delay generic is a vector:

4) The corresponding propagation delay generic (denoting the same input and output ports, condition name, and edge) is identified (see 4.3.2.1.3.14).

 Copyright © 2001 IEEE. All rights reserved.

For each subelement of the biased propagation delay generic, the index of the subelement of the corresponding progragation delay generic is calculated as follows:

Index = (*TbpdGenericIndex)* modulus *(TpdGenericSize*)

The value of the biased propagation delay generic subelement is initialized to the propagation delay generic subelement whose index is computed above.

5) The names of the associated clock and input signals are extracted from the <ClockPort> and <InputPort> portions of the biased propagation delay generic name

6) If there is an internal signal delay generic on the same instance whose name denotes the same <InputPort> and <ClockPort> parts, mark this generic.

For each subelement of the biased propagation delay generic:

7) The indices of the <ClockPort> and the <InputPort> corresponding to the biased propagation delay generic subelement are calculated as follows:

RefPortIndex = (TbpdGenericIndex) / *(TpdGenericSize*)

If the corresponding propagation delay generic is a CrossArc:

TestPortIndex = ((*TbpdGenericIndex)* modulus *(TpdGenericSize*)) / (*OutputPortSize*)

If the corresponding propagation delay generic is a ParallelArc:

TestPortIndex = (TbpdGenericIndex) modulus *(TpdGenericSize*)

8) Mark the subelement of the internal signal delay generic whose index is calculated as follows:

Index = *(TestPortIndex * RefPortSize) + (RefPortIndex)*

9) The value of the biased propagation delay generic subelement is reduced by the value of the marked subelement. If the resulting value is negative, then the value of the subelement is set to 0 ns.

8.2.2.4 Adjustment of propagation delay values

Propagation delay generics are adjusted in two separate steps:

1) All propagation delay timing generics from a clock signal are adjusted

2) Propagation delays which do not correspond to a biased propagation delay generic are adjusted

It is an error if a propagation delay generic is adjusted by more than one internal signal delay.

8.2.2.4.1 Adjustment of clock to output propagation delay values

For each internal clock delay generic:

1) The name of the associated clock signal is extracted from the <ClockPort> portion of the internal clock delay generic name.

Copyright © 2001 IEEE. All rights reserved.

If the internal clock delay generic is a scalar:

2) All propagation delay generics on the same instance are examined. Those generics for which the <InputPort> part of the generic name is the same as the <ClockPort> name are marked.

3) The value of each subelement of each marked generic is reduced by the value of the internal clock delay generic. If the resulting value of any subelement is negative then the value of the subelement is set to 0 ns.

If the internal clock delay generic is a vector:

4) All propagation delay generics on the same instance are examined. Those generics for which the <InputPort> part of the generic name is the same as the <ClockPort> name are marked.

5) For each marked propagation delay generic:

   If the propagation delay generic is a CrossArc, for each subelement of the propagation delay generic, its value is reduced by the value of the internal clock delay generic subelement whose index is computed as follows:

   Index = (TpdGenericIndex) / (OutputPortSize)

   If the propagation delay generic is a ParallelArc, for each subelement of the propagation delay generic,its value is reduced by the value of the internal clock delay generic subelement whose index is computed as follows:

   Index = (*TpdGenericIndex*)

   If the reduced value of any subelement is negative, it is set to 0 ns.

8.2.2.4.2 Adjustment of other propagation delay values

For each internal signal delay timing generic:

1) The names of the associated clock and input signals are extracted from the <ClockPort> and <InputPort> portions of the internal signal delay generic name.

If the internal signal delay generic is a scalar:

2) All propagation delay generics on the instance are examined. If the generic was identified as corresponding to a biased propagation delay generic during the calculation of biased propagation delays then it is not marked. Otherwise, those generics for which the <InputPort> part of the generic name is the same as the <InputPort> name are marked.

3) The value of each subelement of each marked generic is reduced by the value of the internal signal delay generic. If the resulting value of any subelement is negative then the value of the element is set to 0 ns.

If the internal signal delay generic is a vector:

For each subelement of the internal signal delay generic:

 Copyright © 2001 IEEE. All rights reserved.

4) All propagation delay generics on the instance are examined. If the generic was identified as corresponding to a biased propagation delay generic during the calculation of biased propagation delays then it is not marked. Otherwise, those generics for which the <InputPort> part of the generic name is the same as the <InputPort> name are marked

For each marked propagation delay generic:

5) If the propagation delay generic is a CrossArc, mark the subelements whose index is computed as follows:

   Index = (*TisdGenericIndex) * (OutputPortSize) + n*

   where *n* = 0, 1, 2, . . . (*OutputPortSize* - 1)

   If the propagation delay generic is a ParallelArc, mark the subelement whose index is computed as follows:

   Index = *TisdGenericIndex*

   Reduce the value of the marked subelement with the value of the subelement of the internal signal delay generic. If the reduced value is negative, then set it to 0 ns.

8.2.2.5 Adjustment of timing check generics

The timing check generics — setup, hold, recovery, and removal generics — are adjusted in the following steps.

For each internal clock delay generic:

1) The name of the associated clock port is extracted from the <ClockPort> portion of the internal clock delay generic name.

If the internal clock delay generic is a scalar:

2) All setup and recovery generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked. The value of each subelement of each marked generic is increased by the value of the internal clock delay generic.

3) All hold and removal generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked. The value of each subelement of each marked generic is reduced by the value of the internal clock delay generic.

If the internal clock delay generic is a vector:

For each subelement of the clock delay generic:

4) All setup and recovery generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked.

Copyright © 2001 IEEE. All rights reserved.

For each such marked setup/recovery generic, each subelement is increased by the value of the subelement of the clock delay generic whose index is computed as follows:

Index = (*TsetupRecoveryGenericIndex)* modulus *(RefPortSize)*

5) All hold and removal generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked.

   For each such marked hold/removal generic, each subelement is decreased by the value of the subelement of the clock delay generic whose index is computed as follows:

   Index = TholdRemovalGenericIndex) modulus *(RefPortSize*)

For each internal signal delay generic:

1) The names of the associated clock and input ports are extracted from the <ClockPort> and <InputPort> portions of the internal signal delay generic name.

If the internal signal delay generic is a scalar:

2) All setup and recovery generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name and the <TestPort> part is the same as the <InputPort> name are marked. The value of each subelement of each marked generic is reduced by the value of the internal signal delay generic. If the resulting value of any subelement is negative then the value of the subelement is set to 0 ns.

3) All hold and removal generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name and the <TestPort> part is the same as the <InputPort> name are marked. The value of each subelement of each marked generic is increased by the value of the internal signal delay generic.

If the internal signal delay generic is a vector:

4) All setup and recovery generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name and the <TestPort> part is the same as the <InputPort> name are marked.

   For each marked setup/recovery generic, the value of each subelement is reduced by the value of the subelement of the internal signal delay generic whose index is computed as follows:

   Index = *TisdGenericIndex*

   If the resulting value of any subelement is negative then the value of the subelement is set to 0 ns.

5) All hold and removal generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name and the <TestPort> part is the same as the <InputPort> name are marked.

   For each marked hold/removal generic, the value of each subelement of each marked generic is increased by the value of the subelement of the internal signal delay generic whose index is computed as follows:

   Index = *TisdGenericIndex*

 Copyright © 2001 IEEE. All rights reserved.

For each internal signal delay generic:

1) The names of the associated clock and input ports are extracted from the <ClockPort> and <InputPort> portions of the internal signal delay generic name.

2) All hold and removal generics on the same instance are examined. Those generics for which the <ReferencePort> part of the generic name is the same as the <ClockPort> name are marked. If the value of any subelement of the marked generic is negative, it is set to 0 ns.

Copyright © 2001 IEEE. All rights reserved.

9. Delay selection

The VITAL ASIC modeling specification supports propagation delay path selection and signal output scheduling in both sequential and concurrent contexts. These activities are performed by a number of predefined procedures provided for use by VITAL Level 1 models. The predefined procedures are:

— The VITAL path delay procedures, for use in the path delay section of a VITAL process
— The VITAL concurrent primitives, for use in concurrent procedure calls

9.1 VITAL delay types and subtypes

Package VITAL_Timing defines a number of delay types and subtypes which support the specification and selection of simple delay values as well as delay values corresponding to the transitions between any of the states '0', '1', 'X', and 'Z'. A delay whose value may vary according to the nature of a transition is called a *transition dependent delay*. A delay with no such dependence is a *simple delay*.

```
type VitalTransitionType is (tr01, tr10, tr0z, trz1, tr1z, trz0, tr0x, trx1, tr1x, trx0, trxz, trzx);

subtype VitalDelayType is Time;
type VitalDelayType01   is array (VitalTransitionType range tr01 to tr10) of Time;
type VitalDelayType01Z  is array (VitalTransitionType range tr01 to trz0)  of Time;
type VitalDelayType01ZX is array (VitalTransitionType range tr01 to trzx)  of Time;

type VitalDelayArrayType is array (NATURAL range <>) of VitalDelayType;
type VitalDelayArrayType01 is array (NATURAL range <>) of VitalDelayType01;
type VitalDelayArrayType01Z is array (NATURAL range <>) of VitalDelayType01Z;
type VitalDelayArrayType01ZX is array (NATURAL range <>) of VitalDelayType01ZX;
```

A transition dependent delay is represented by a value of a *transition dependent delay type*. Similarly, a simple delay is represented by a value of a *simple delay type*. There are a number of different transition dependent delay types representing different subsets of transitions. Each kind of simple or transition dependent delay type has both scalar and vector forms. The vector forms represent delay values corresponding to one or more vector ports for which the delay(s) associated with each bit may be different.

A value of a transition dependent delay type associates a (possibly) different delay value with each transition in a set of transitions. The value takes the form of an array of delay times, indexed by transition values. Each element delay value is associated with the transition corresponding to its index position. The transition dependent delay types are VitalDelayType01, VitalDelayType01Z, VitalDelayType01ZX, VitalDelayArrayType01, VitalDelayArrayType01Z, and VitalDelayArrayType01ZX. The first three are scalar forms, and the last three are vector forms.

A value of a simple delay type is a single delay value, or a vector of single delay values corresponding to one or more vector ports. Although the vector form of a simple delay is an array, the delays that it represents are not associated with transitions. The simple delay types and subtypes include Time, VitalDelayType, and VitalDelayArrayType. The first two are scalar forms, and the latter is the vector form.

The simple delay types and subtypes and the transition dependent delay types comprise the set of *VITAL delay types and subtypes*. No other type or subtype is considered to be a VITAL delay type or subtype.

 Copyright © 2001 IEEE. All rights reserved.

9.2 Transition dependent delay selection

Delay selection for a particular signal may be based upon the new and previous values of the signal; this selection mechanism is called *transition dependent delay selection*. Transitions between the previous and new values are described by enumeration values of the predefined type VitalTransitionType. The following table describes the delay selection for a set of previous and current values:

Table 7—Transition Dependent Delay Selection

Previous value	New value	Delay selected for VitalDelayType	Delay selected for VitalDelayType01	Delay selected for VitalDelayType01Z
'0'	'1'	Delay	Delay(tr01)	Delay(tr01)
'0'	'Z'	Delay	Delay(tr01)	Delay(tr0Z)
'0'	'X'	Delay	Delay(tr01)	Min(Delay(tr01), Delay(tr0Z))
'1'	'0'	Delay	Delay(tr10)	Delay(tr10)
'1'	'Z'	Delay	Delay(tr10)	Delay(tr1Z)
'1'	'X'	Delay	Delay(tr10)	Min(Delay(tr10), Delay(tr1Z))
'Z'	'0'	Delay	Delay(tr10)	Delay(trZ0)
'Z'	'1'	Delay	Delay(tr01)	Delay(trZ1)
'Z'	'X'	Delay	Min(Delay(tr10), Delay(tr01))	Min(Delay(trZ1), Delay(trZ0))
'X'	'0'	Delay	Delay(tr10)	Max(Delay(tr10), Delay(trZ0))
'X'	'1'	Delay	Delay(tr01)	Max(Delay(tr01), Delay(trZ1))
'X'	'Z'	Delay	Max(Delay(tr10), Delay(tr01))	Max(Delay(tr1Z), Delay(tr0Z))

9.3 Glitch handling

A *glitch* occurs when a new transaction is scheduled at an absolute time which is greater than the absolute time of a previously scheduled pending event which results in a preemptive behavior. The preemptive behavior can be either positive or negative. Glitch handling in a VITAL Level 1 model is incorporated into the signal scheduling mechanism.

The VITAL ASIC modeling specification supports four modes of signal scheduling. These modes are represented by the enumeration values of the predefined VITAL type VitalGlitchKindType:

type VitalGlitchKindType **is** (OnEvent, OnDetect, VitalInertial, VitalTransport);

The VitalInertial and VitalTransport modes are identical to the inertial and transport modes of VHDL. The OnEvent and OnDetect modes are special modes for glitch handling. In the OnEvent mode, a glitch causes an 'X' value to be scheduled on the output at the time when the scheduled event was to occur. In the OnDetect mode, a glitch causes an 'X' value to be scheduled on the output at the time of glitch detection. The default glitch behavior is positive preemption. If the NegPreemptOn parameter is True in VitalPathDelay procedures, negative preemption behavior is enabled.

Copyright © 2001 IEEE. All rights reserved.

Example:

Consider a simple buffer with non-symmetrical delays. The outputs from the buffer corresponding to various glitch modes are shown in figure 9-1 for both positive and negative preemptive behaviors. "A" is the first scheduled event and "B" is the second scheduled event.

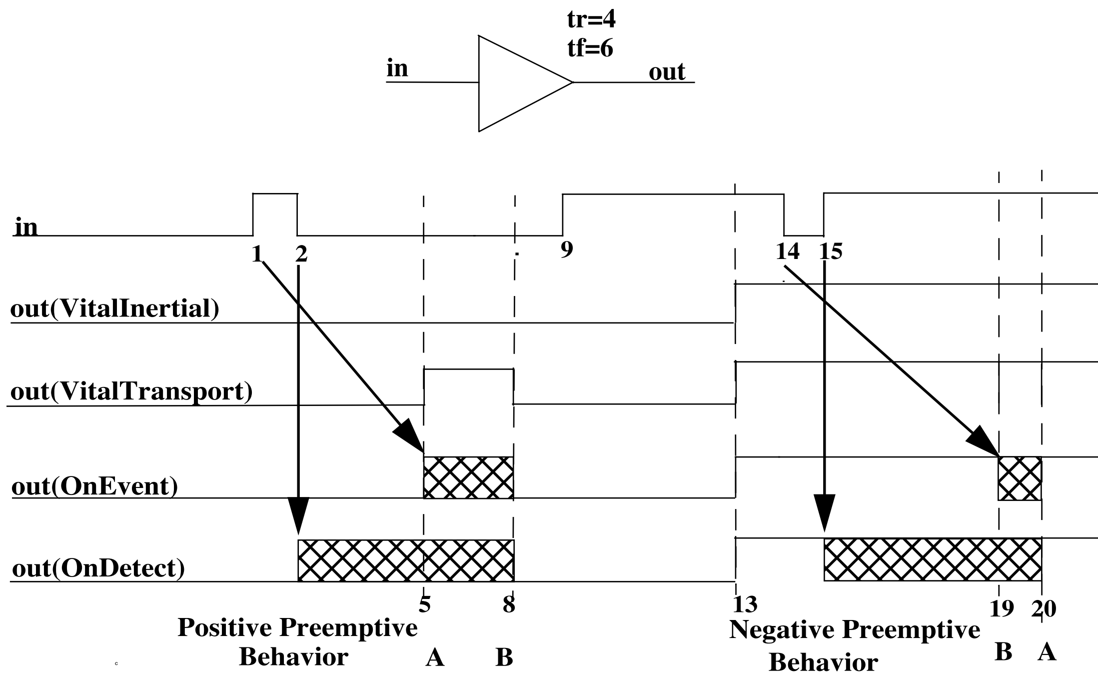


Figure 8—VITAL delay modes

9.4 Path delay procedures

Signal scheduling inside a VITAL Level 1 process can be performed by one of the predefined path delay procedures VitalPathDelay, VitalPathDelay01, and VitalPathDelay01Z. Each of these procedures provides the following capabilities:

- Transition dependent path delay selection
- User controlled glitch detection, 'X' generation, and violation reporting
- Scheduling of the computed values on the specified signal

The information about all the relevant paths to a particular output is specified by using the Paths parameter. The following record structure is used to convey information about an input to output path:

```
type VitalPath01Type is record
    InputChangeTime    : TIME;              -- timestamp for path input signal
    PathDelay          : VitalDelayType01;  -- delay for this path
    PathCondition      : BOOLEAN;           -- path sensitize condition
end record;
```


Copyright © 2001 IEEE. All rights reserved.

Selection of the appropriate path delay begins with the selection of candidate paths. The candidate paths are selected by identifying the paths for which the PathCondition is True. If there is a single candidate path, its delay is the one selected. If there is more than one candidate path, the shortest delay (accounting for the InputChangeTime) is selected using transition dependent delay selection. If RejectFastPath parameter is True and the candidate path delay selected is less than the non-matured path delay and the computed value is the same as the non-matured scheduled value, no path delay is selected. If there are no candidate paths then the delay used by the path delay procedure is either specified by the DefaultDelay parameter or TIME'HIGH. If the IgnoreDefaultDelay parameter is True the delay selected is TIME'HIGH; othewise the delay is specified by the DefaultDelay parameter.

9.4.1 VitalPathDelay and VitalPathDelay01

The VitalPathDelay and VitalPathDelay01 procedures schedule path delays on signals for which the transition to 'Z' is not important. These procedures are distinguished from one another by the type of delay values that they accept. The procedure VitalPathDelay is defined for simple path delays of type VitalDelayType. Procedure VitalPathDelay01 is defined for transition dependent path delays of type VitalDelayType01 (rise/fall delays).

Example:

```
VitalPathDelay01(
    OutSignal        => QN,      -- signal being scheduled
    OutSignalName => "QN",    -- name of the signal
    OutTemp          => QN_zd, -- new signal value to be scheduled
   Paths              => (          -- one path data for each input affecting the output
        0   =>  (InputChangeTime =>  CLK_ipd'LAST_EVENT,  --  1st   input  which  affects
output
                  PathDelay           => tpd_CLK_QN,
                  Condition            => (PN_Ipd = '0' and CN_ipd = '1' ),
       1    =>  (InputChangeTime =>  PN_ipd'LAST_EVENT,    --  2nd  input  which  affects
output
                  PathDelay           => tpd_PN_QN,
                  Condition            => (CN_ipd = '1')),
       2    =>  (InputChangeTime => CN_ipd'LAST_EVENT,  -- 3rd input which affects output
                  PathDelay           => tpd_CN_QN,
                  Condition            => (PN_ipd = '0'))),
    GlitchData       => GlitchData_QN,
   DefaultDelay     => VitalZeroDelay01,-- Delay to be used if all path condition are FALSE
    Mode              => OnEvent,    -- Mode for Glitch processing
    MsgOn            => TRUE,        -- Message control on glitch
    XOn               => TRUE,        -- X-generation on glitch
    MsgSeverity     =>  ERROR,
    NegPreemptOn   =>  TRUE,-- Enable negative preemptive glitch handling
    IgnoreDefaultDelay =>  TRUE,  -- Ignore use of the default delay
    RejectFastPath   =>   TRUE);    -- Enable rejection of fast signal path
```

9.4.2 VitalPathDelay01Z

Procedure VitalPathDelay01Z schedules path delays on signals for which the transition to or from 'Z' is important (e.g., modeling of tri-state drivers). In addition to the basic capabilities provided by all path delay procedures,VitalPathDelay01Z performs result mapping of the output value (using the value specified by the actual associated with the OutputMap parameter) before scheduling this value on the signal. This result mapping is performed after transition dependent delay selection but before scheduling the final output.

Copyright © 2001 IEEE. All rights reserved.

Example:

```
VitalPathDelay01Z(
  OutSignal       => Q,        -- signal being scheduled
  OutSignalName => "Q",       -- name of the signal
  OutTemp         => Q_zd,   -- new signal value
  Paths           => (         -- one path data for each input affecting the output
  0   =>      (InputChangeTime => D_ipd'LAST_EVENT,          -- 1st  input which affects
output
                PathDelay          => tpd_D_Q,
                Condition          => (Enable = '0')),
  1   =>      (InputChangeTime => Enable_ipd'LAST_EVENT, -- 2nd input which affects
output
                PathDelay          => tpd_Enable_Q,
                Condition          => (Enable = '1'))),
  GlitchData      =>  GlitchData_Q,
  MsgOn           =>  TRUE,
  XOn             =>  TRUE,
  Mode            =>  OnEvent,
  MsgSeverity     =>  ERROR,
  NegPreemptOn   =>  TRUE,      -- Enable negative preemptive glitch handling
  IgnoreDefaultDelay =>  TRUE,   -- Ignore use of the default delay
  RejectFastPath  =>  TRUE,      -- Enable rejection of fast signal path;
  OutputMap       =>  "UX01WHLHX"); -- Pullup behavior .
```

9.5 Delay selection in VITAL primitives

In addition to functional computation, the VITAL primitive procedures perform delay selection, glitch handling, and signal scheduling. The delay selection mechanism in the primitives is different from that used in the path delay procedures.

The delay selection algorithm used by the VITAL primitive procedures is based on the following selection criteria:

— If the new output value is dependent on multiple input values, the delay selected is the maximum of the delays from the dependent inputs.
— If the new output value is determined by either of the input values, the delay selected is the minimum of the delays from these inputs.

Delay selection in VITAL primitive procedures is accomplished by maintaining separate output times from each input signal and then selecting the appropriate output delay based on the above selection criteria. The new value is scheduled on the output using the selected delay.

Control of glitch handling is provided through a formal parameter.

Example:

Let
Ti0 be the time when the output will change based on a falling input
Ti1 be the time when the output will change based on a rising input

For an AND primitive,
An output going to a '1' value will be scheduled after the maximum of Ti1 times for each input

 Copyright © 2001 IEEE. All rights reserved.

An output going to a ‘0’ value will be scheduled after the minimum of Ti0 times for each input

However, for a NAND primitive,
An output going to a ‘1’ value will be scheduled after the minimum of Ti0 times for each input
An output going to a ‘0’ value will be scheduled after the maximum of Ti1 times for each input

Similarly, for an OR primitive.
An output going to a ‘1’ value will be scheduled after the minimum of Ti1 times for each input
An output going to a ‘0’ value will be scheduled after the maximum of Ti0 times for each input

9.6 VitalExtendToFillDelay

The function VitalExtendToFillDelay is a utility that provides a set of six transition dependent delay values, even though fewer delay values may have been explicitly provided.

Example:

```
CONSTANT tpd_Input_Output : VitalDelayType01;        -- This variable holds 2 delay values
VARIABLE tpd_Control_Output: VitalDelayType01Z;      -- This variable holds 6 delay values
    ...
tpd_Control_Output := VitalExtendToFillDelay(tpd_Input_Output);
```

Copyright © 2001 IEEE. All rights reserved.

10. The Level 1 Memory specification

The VITAL Level 1 Memory specification is a set of modeling rules which constrains the descriptions of ASIC memory models. This facilitates the optimization of the set-up and execution of the models, leading to higher levels of simulation performance.

A VITAL Level 1 Memory model description defines an ASIC memory in terms of its declaration, functionality, corruption handling, wire delay propagation, timing constraints, output delay selection and scheduling.

10.1 The VITAL Level 1 Memory attribute

A VITAL Level 1 Memory architecture is identified by its decoration with the VITAL_Level1_Memory attribute, which indicates an intention to adhere to the Level 1 Memory specification.

VITAL_Level1_Memory_attribute_specification ::= attribute_specification

A VITAL Level 1 Memory architecture shall contain a specification of the VITAL_Level1_Memory attribute corresponding to the declaration of that attribute in package VITAL_Memory. The expression in the VITAL_Level1_Memory attribute specification shall be the Boolean literal True.

Example:

attribute VITAL_Level1_Memory **of** VitalCompliantMemArchitecture : **architecture is** True;

10.2 The VITAL Level 1 Memory architecture body

A Level 1 Memory architecture body defines the body of a VITAL Level 1 Memory design entity.

VITAL_Level1_Memory_architecture_body ::=
    architecture identifier **of** *entity*_name **is**
        VITAL_Level1_Memory_architecture_declarative_part
    begin
        VITAL_Level1_Memory_architecture_statement_part
    end [*architecture*_simple_name] ;

A VITAL Level 1 Memory architecture shall adhere to the VITAL Level 0 specification, except for the declaration of the VITAL_Level0 attribute.

The entity associated with a VITAL Level 1 Memory architecture shall be a VITAL Level 0 entity. Together, these design units comprise a *Level 1 Memory design entity*.

The only signals that shall be referenced in a VITAL Level 1 Memory design entity are entity ports and internal signals. References to global signals, shared variables and signal-valued attributes are not allowed. Each signal declared in a VITAL Level 1 Memory design entity shall have at most one driver.

The use of subprogram calls and operators in a VITAL Level 1 Memory architecture is limited. The only operators or subprograms that shall be invoked are those declared in package Standard, package Std_Logic_1164, or the VITAL standard packages. Formal sub-element associations and type conversions are prohibited in the associations of a subprogram call.

 Copyright © 2001 IEEE. All rights reserved.

10.3 The VITAL Level 1 Memory architecture declarative part

The VITAL Level 1 Memory architecture declarative part contains declarations of items that are available for use within the VITAL Level 1 Memory architecture.

```
VITAL_Level1_Memory_architecture_declarative_part ::=
    VITAL_Level1_Memory_attribute_specification
    { VITAL_Level1_Memory_block_declarative_item }

VITAL_Level1_Memory_block_declarative_item ::=
      constant_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | VITAL_memory_internal_signal_declaration
```

10.3.1 VITAL memory internal signals

A signal that is declared in the declarative part of a VITAL memory architecture is an *internal signal*.

```
VITAL_memory_internal_signal_declaration ::=
    signal identifier_list : type_mark [ index_constraint ] [ := expression ] ;
```

The type mark in the declaration of an internal signal shall denote the standard logic type Std_Ulogic or Std_Logic_Vector. The index constraint in the signal declaration shall adhere to the rules specified in 10.4.1.1.1.

10.4 The VITAL Level 1 Memory architecture statement part

The statement part of a VITAL Level 1 Memory architecture is a set of one or more concurrent statements that perform specific VITAL activities.

```
VITAL_Level1_Memory_architecture_statement_part ::=
    VITAL_Level1_Memory_concurrent_statement {
VITAL_Level1_Memory_concurrent_statement }

VITAL_Level1_Memory_concurrent_statement ::=
      VITAL_wire_delay_block_statement
    | VITAL_negative_constraint_block_statement
    | VITAL_memory_process_statement
    | VITAL_memory_output_drive_block_statement
```

A VITAL Level 1 Memory architecture shall contain at most one wire delay block statement (see 6.4.1).

If the entity associated with a VITAL Level 1 Memory architecture declares one or more timing generics representing internal clock or internal signal delay, negative constraints are in effect. The VITAL Level 1 Memory architecture shall contain exactly one negative constraint block to compute the associated signal delays (see 6.4.2).

A VITAL Level 1 Memory architecture shall contain exactly one VITAL memory process statement. A VITAL Level 1 Memory architecture shall contain at most one memory output drive block statement.

Copyright © 2001 IEEE. All rights reserved.

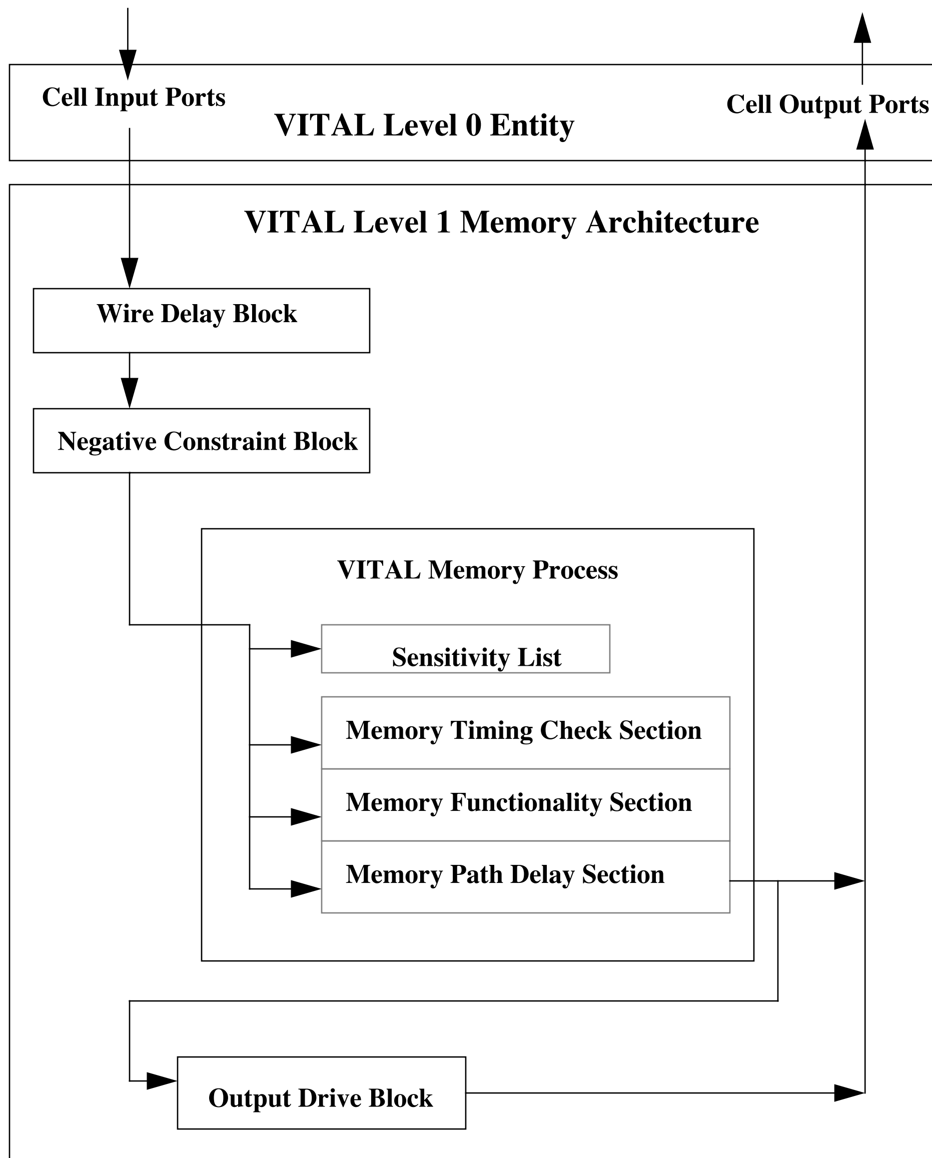


Figure 9—Structure of a VITAL Level 1 Memory model

10.4.1 VITAL memory process statement

A VITAL memory process statement is a key building block of a VITAL Level 1 Memory architecture. It is a mechanism for modeling memory specific timing constraints, functionality, and path delays.

VITAL_memory_process_statement ::=
  *process*_label :
  process (sensitivity_list)
    VITAL_memory_process_declarative_part
  begin
    VITAL_memory_process_statement_part
  end process [*process*_label] ;

 Copyright © 2001 IEEE. All rights reserved.

The label of the VITAL memory process shall be the name MemoryBehavior.

A VITAL memory process statement shall have a sensitivity list. The sensitivity list shall contain the longest static prefix of every signal name that appears as a primary in a context in which the value of the signal is read. These are the only signal names that the sensitivity list may contain.

10.4.1.1 VITAL memory process declarative part

A VITAL memory process declarative part is restricted to a few kinds of declarations.

```
VITAL_memory_process_declarative_part ::=
    { VITAL_memory_process_declarative_item }

VITAL_memory_process_declarative_item ::=
      constant_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | VITAL_variable_declaration
    | VITAL_memory_variable_declaration
```

NOTE—The declaration of the memory object shall be done through a call to the VitalDeclareMemory function.

10.4.1.1.1 VITAL memory variables

VITAL memory variables fall into two classes that are distinguished by the manner in which their members are used. A VITAL *memory restricted variable* is a variable that is required to store persistent data for private use by certain procedures in the VITAL memory package. The proper functioning of these procedures requires that the content of the variable not be modified separately by the model itself, hence the use of a memory restricted variable within a VITAL Level 1 Memory architecture is strictly limited. A VITAL memory variable that is not a memory restricted variable is called a *memory unrestricted variable*.

VITAL_memory_variable_declaration ::=
    variable identifier_list : type_mark [index_constraint] [:= expression] ;

The type of a memory unrestricted variable shall be one of the standard logic types Std_ulogic, Std_logic_vector, Boolean, Time or one of the types defined in the package VITAL_Memory.

For a VITAL memory variable that is of vector form, the index constraint range shall be a descending range with the right index of the variable being 0.

10.4.1.1.1.1 VITAL memory restricted variables

A VITAL memory restricted variable is identified by its association with a *restricted formal parameter*. This single association is the only use permitted for a VITAL memory restricted variable. Certain formal parameters of some procedures in the VITAL memory package are designated as *restricted formal parameters*. These formal parameters and their corresponding procedures are:

Copyright © 2001 IEEE. All rights reserved.

Table 8—VITAL Memory Restricted Variables

Restricted Formal parameter	VITAL Memory procedures
MemoryData	VitalMemoryTable VitalMemoryViolation VitalMemoryCrossPorts
PrevControls	VitalMemoryTable
PrevDataInBus	VitalMemoryTable
PrevAddressBus	VitalMemoryTable
PrevEnableBus	VitalMemoryTable
PortFlag	VitalMemoryTable VitalMemoryViolation
PortFlagArray	VitalMemoryTable
AddressValue	VitalMemoryTable VitalMemoryViolation
ScheduleData	VitalMemoryInitPathDelay VitalMemoryAddPathDelay VitalMemorySchedulePathDelay
ScheduleDataArray	VitalMemoryInitPathDelay VitalMemoryAddPathDelay VitalMemorySchedulePathDelay
InputChangeTime	VitalMemoryAddPathDelay
InputChangeTimeArray	VitalMemoryAddPathDelay
TimingData	VitalMemorySetupHoldCheck
PeriodPulseData	VitalMemoryPeriodPulseCheck

The actual part in the association of a restricted formal parameter shall be the simple name of a VITAL memory restricted variable of the same type or an aggregate of variables of the same type mark as the formal parameter.

Certain restrictions are placed on the declaration of a VITAL memory restricted variable. The type mark in the memory restricted variable declaration shall denote the type or subtype denoted by the type mark in the corresponding restricted formal parameter declaration. If the declaration of the memory restricted variable contains an initial value expression then that expression shall take one of the following forms:

- It can be the name of a constant declared in the VITAL_Memory package.
- It can a call to VitalDeclareMemory function specified in the VITAL_Memory package. The actual parameter part in such a function call shall be a locally static expression.

10.4.1.2 VITAL memory process statement part

The VITAL memory process statement part consists of statements that describe memory timing constraint checks, memory functionality and memory path delay selection.

 Copyright © 2001 IEEE. All rights reserved.

VITAL_memory_process_statement_part ::=
[VITAL_memory_timing_check_section]
[VITAL_memory_functionality_section]
[VITAL_memory_path_delay_section]

These statements are grouped into three distinct sections, describing the memory timing check specification, memory functional specification and the memory path delay specification. A VITAL memory process shall include at least one of these sections.

10.4.1.2.1 VITAL memory timing check section

The VITAL memory timing check section performs timing constraint checks through the invocation of predefined timing check procedures defined in the VITAL_Timing and VITAL_Memory packages. Timing checks that can be performed in this section include setup/hold checks, recovery/removal checks, period/pulsewidth checks and skew checks.

VITAL_memory_timing_check_section ::=
if VITAL_memory_timing_check_condition **then**
{ VITAL_memory_timing_check_statement }
end if ;

VITAL_memory_timing_check_condition ::= *generic*_simple_name

VITAL_memory_timing_check_statement ::= procedure_call_statement

The VITAL memory timing check condition shall be a simple name denoting a TimingChecksOn control generic that shall be declared in the entity.

The VITAL memory timing check statement is a procedure call statement that invokes any of the following timing check procedures declared in either VITAL_Timing or VITAL_Memory packages:

— The procedures declared in package VITAL_Timing are VitalSetupHoldCheck, VitalRecoveryRemovalCheck, VitalPeriodPulseCheck, VitalInPhaseSkewCheck, and VitalOutPhaseSkewCheck .
— The procedures declared in package VITAL_Memory are VitalMemorySetupHoldCheck, VitalMemoryPeriodPulseCheck.

Each of these procedures performs the specified constraint check and returns a parameter value or values in the formal argument Violation, indicating whether or not a constraint violation occurred (see 8.1, 12.6). These values are considered to be the output(s) of the timing check section and shall be used to implement memory corruption behavior.

A timing check section is the only context in which a call to a timing check procedure is allowed. The actual parameter part of a timing check procedure call shall satisfy the following requirements:

— The actual part associated with a formal parameter representing a signal name shall be a locally static name.
— The actual part associated with the formal parameter HeaderMsg shall be a globally static expression.
— The actual part associated with a formal parameter of the standard type Time shall be a globally static expression.
— The actual part associated with a formal parameter XOn shall be a locally static expression or a simple name denoting the control generics XOn or XOnChecks.

Copyright © 2001 IEEE. All rights reserved.

— The actual part associated with a formal parameter MsgOn shall be a locally static expression or a simple name denoting the control generics MsgOn or MsgOnChecks.
— An actual part associated with the formal parameter TestSignalName, RefSignalName, RefTransition, Signal1Name, Signal2Name, MsgFormat or ArcType shall be a locally static expression.
— An actual part associated with the formal parameter SetupHigh, SetupLow, HoldHigh, HoldLow, RefDelay, TestDelay, Period, PulseWidthHigh, PulseWidthLow shall satisfy the constraint as defined for timing generics (see 4.3.2.1.2).
— A function call or operator in an actual part shall invoke a function or operator that is defined in package Standard, package Std_Logic_1164, or package VITAL_Timing.

Timing checks shall be independent of one another. It is an error for a variable that is associated with a timing check Violation parameter to appear in another timing check statement.

NOTE—Although the actual part associated with Violation parameter cannot be used more than once within the VITAL timing check section it is allowed to be used in the context of any other section such as functionality section.

10.4.1.2.2 VITAL memory functionality section

The VITAL memory functionality section describes the behavior of the memory.

VITAL_memory_functionality_section ::=
{ VITAL_variable_assignment_statement | VITAL_memory_procedure_call_statement }

VITAL_variable_assignment_statement ::=
VITAL_target := expression ;

VITAL_target ::= *memory_unrestricted_variable_name*

The function of a memory model is specified in terms of variable assignment statements and VITAL memory procedure call statements.

A VITAL memory procedure call statement in the memory functionality section shall invoke one or more of the predefined procedures:

— VitalMemoryTable, VitalMemoryViolation and VitalMemoryCrossPorts that are defined in the VITAL_Memory package (see Clause 11). The memory functionality section is the only context in which a call to these procedures is allowed.
— VitalStateTable that is defined in package VITAL_Primitives (see 6.4.3.2.2 and 7.3.4).

The use of these procedures shall adhere to the following rules :

— There shall be at least one call to VitalMemoryTable in the VITAL memory functionality section.
— If present, all the calls to the VitalMemoryViolation procedure shall follow after the last call to VitalMemoryTable procedure.
— If present, all the calls to the VitalMemoryCrossPorts procedure shall follow after the last call to the VitalMemoryViolation procedure.

Certain restrictions are placed on a VITAL variable assignment statement. The target shall be an unrestricted variable that is denoted by a locally static name, and the right-hand side expression shall be such that every primary in the right-hand side expression is one of the following:

— A globally static expression
— A name denoting a variable, a port, or an internal signal

 Copyright © 2001 IEEE. All rights reserved.

— A function call invoking a standard logic function, a VITAL primitive, or the function VITALTruthTable
— An aggregate, or a qualified expression whose operand is an aggregate
— A parenthesized expression

See 6.4.3.2.2 and 7.3.3 for details on VITALTruthTable usage.

10.4.1.2.2.1 VitalMemoryTable argument restrictions

The actual parameter part of the procedure call VitalMemoryTable shall satisfy the following requirements:

— The actual part associated with the formal parameter MemoryData representing a variable shall be a static name and furthermore, shall only be the output of the VitalDeclareMemory function.
— An actual part associated with the MemoryTable formal parameter in a call to VitalMemoryTable procedure shall be a constant that is not a deferred constant. Furthermore, the value expression of that constant shall be a positional aggregate formed using only locally static expressions or nested aggregates of this form.
— The constraint on the variable associated with the PreviousDataInBus parameter shall match that on the actual associated with the DataInBus parameter.
— The constraint on the variable associated with the PreviousControls parameter shall match that on the actual associated with the Controls parameter.
— The constraint on the variable associated with the PreviousEnableBus parameter shall match that on the actual associated with the EnableBus parameter.
— The constraint on the variable associated with the PreviousAddressBus parameter shall match that on the actual associated with the AddressBus parameter.
— An actual part associated with the formal parameter PortName, PortType, PortFlag, PortFlagArray, HeaderMsg, MsgSeverity shall be a locally static expression.
— The actual part associated with the formal parameter MsgOn shall be a locally static expression or a simple name denoting the control generic MsgOn.

10.4.1.2.2.2 VitalMemoryViolation argument restrictions

The actual parameter part of the procedure call VitalMemoryViolation shall satisfy the following requirements:

— The actual part associated with the formal parameter MemoryData representing a variable shall be a static name and furthermore, shall only be the output of the VitalDeclareMemory function.
— An actual part associated with the ViolationTable formal parameter in a call to VitalMemoryViolation procedure shall be a constant that is not a deferred constant. Furthermore, the value expression of that constant shall be a positional aggregate formed using only locally static expressions or nested aggregates of this form.
— The actual part associated with the formal parameter AddressValue representing a variable shall be a static name. This variable shall only be the actual part of AddressValue in a call to VitalMemoryTable associated with this call to VitalMemoryViolation.
— The actual part associated with the formal parameter PortFlag shall be an aggregate of scalar port flags. Each member of the aggregate shall only be the actual part of PortFlag in a call to VitalMemoryTable associated with this call to VitalMemoryViolation.
— The actual part associated with the formal parameter PortFlagArray shall be an aggregate of vector port flags. Each member of the aggregate shall only be the actual part of PortFlagArray in a call to VitalMemoryTable associated with this call to VitalMemoryViolation.
— The actual part associated with the formal parameter ViolationFlags shall be an aggregate of scalar violation variables.

Copyright © 2001 IEEE. All rights reserved.

— The actual part associated with the formal parameter ViolationFlagsArray shall be an aggregate of vector violation variables. The actual part associated with the formal parameter ViolationSizesArray shall be an aggregate of positive integers. Each member of the aggregate shall match the size of the corresponding vector violation variable in ViolationFlagsArray.
— An actual part associated with the formal parameter PortName, HeaderMsg shall be a locally static expression.
— An actual part associated with the formal parameter PortType shall be a simple name of the type VitalPortType.
— The actual part associated with a formal parameter MsgOn shall be a locally static expression or a simple name denoting the control generic MsgOn.

10.4.1.2.2.3 VitalMemoryCrossPorts argument restrictions

The actual parameter part of the procedure call VitalMemoryCrossPorts shall satisfy the following requirements:

— The actual part associated with the formal parameter MemoryData representing a variable shall be a static name and furthermore, shall only be the output of the VitalDeclareMemory function.
— The actual part associated with the formal parameter SamePortFlag shall only be the actual part of PortFlag in a call to VitalMemoryTable associated with this call to VitalMemoryCrossPorts.
— The actual part associated with the formal parameter SamePortAddressValue representing a variable shall be a static name. This variable shall only be the actual part of AddressValue in a call to VitalMemoryTable associated with this call to VitalMemoryCrossPorts.
— The actual part associated with the formal parameter CrossPortFlagArray shall be an aggregate of PortFlagArray. Each member of the aggregate shall only be the actual part of PortFlagArray in a call to VitalMemoryTable associated with this call to VitalMemoryCrossPorts.
— The actual part associated with the formal parameter CrossPortAddressArray shall be an aggregate of AddressValue. Each member of the aggregate shall only be the actual part of AddressValue in a call to VitalMemoryTable associated with this call to VitalMemoryCrossPorts.
— An actual part associated with the formal parameter PortName, HeaderMsg shall be a locally static expression.
— An actual part associated with the formal parameter CrossPortMode shall be a simple name of the type VitalCrossPortModeType.
— The actual part associated with a formal parameter MsgOn shall be a locally static expression or a simple name denoting the control generic MsgOn.

10.4.1.2.3 VITAL memory path delay section

The VITAL memory path delay section performs propagation delay selection and output signal scheduling in VITAL Level1 Memory models. The VITAL memory path delay section drives ports or internal signals using appropriate delay values. Path delay selection and scheduling is modeled through procedure call statements which use the pre-defined path delay procedures — VitalMemoryInitPathDelay, VitalMemoryAddPathDelay and VitalMemorySchedulePathDelay defined in the package VITAL_Memory. The VITAL memory path delay section is the only context in which a call to these procedures is allowed.

The procedure calls to the VITAL memory path delay procedures corresponding to an output port(s) or internal signal(s) shall satisfy the following requirements:

— There shall be exactly one call each to the VitalMemoryInitPathDelay and VitalMemorySchedulePathDelay procedures. There shall be at least one call to the VitalMemoryAddPathDelay procedure.

 Copyright © 2001 IEEE. All rights reserved.

— All calls to the VitalMemoryAddPathDelay procedure shall follow the VitalMemoryInitPathDelay procedure. The call to the VitalMemorySchedulePathDelay procedure shall follow after the last call to the VitalMemoryAddPathDelay procedure.

NOTE— Though the above sequence is defined per ouput, it is allowed to group a set of VitalMemoryInitPathDelay calls. The same rule applies to VitalMemoryAddPathDelay and VitalMemorySchedulePathDelay procedure calls.

The actual part of a VitalMemoryAddPathDelay procedure call shall satisfy the following requirements:

— For a particular output, the actual part associated with ScheduleData or ScheduleDataArray shall be the actual part of ScheduleData or ScheduleDataArray respectively, in a call to VitalMemoryInitPathDelay procedure associated with this call to VitalMemoryAddPathDelay.
— The actual part associated with InputChangeTime or InputChangeTimeArray shall either be a Last_Event attribute or a locally static expression.
— The actual part associated with PathDelay or PathDelayArray shall satisfy the constraints as defined for timing generics (see 4.3.2.1.2).
— An actual part associated with the formal parameter ArcType shall be a simple name of the type VitalMemoryArcType.

The actual parameter part of a VitalMemorySchedulePathDelay procedure call shall satisfy the following requirements:

— For a particular output, the actual part associated with ScheduleData or ScheduleDataArray shall be the actual part of ScheduleData or ScheduleDataArray respectively, in a call to VitalMemoryInitPathDelay procedure associated with this call to VitalMemorySchedulePathDelay.
— The actual part associated with the formal parameter OutSignal shall be a locally static name.
— An actual part associated with the formal parameter OutSignalName or OutputMap shall be a locally static expression.

10.4.2 VITAL memory output drive block

The VITAL memory output drive block is used to propagate internal signals to the output ports. This block can also be used to model output enable behavior.

VITAL_output_drive_block_statement ::=
  *block*_label :
  block
  begin
    VITAL_output_drive_block_statement_part
  end block [*block*_label] ;

VITAL_output_drive_block_statement_part ::=
  { VITAL_primitive_concurrent_procedure_call
  | concurrent_signal_assignment_statement }

The label of a VITAL memory output drive block shall be the name OutputDrive.

The VITAL primitive concurrent procedure call shall only invoke one of the following primitives defined in package VITAL_Primitives — VitalBUF, VitalBUFIF0, VitalBUFIF1, VitalIDENT.

Copyright © 2001 IEEE. All rights reserved.

The actual parameter part of a primitive procedure call shall satisfy the following requirements:

— An actual part associated with a formal parameter of class SIGNAL shall be a static name. The actual part associated with the formal parameter Data (for VitalBUFIF0, VitalBUFIF1) or the formal parameter a (for VitalBUF,VitalIdent) shall be an output of the VITAL memory path delay section.
— An actual part associated with a formal parameter of class CONSTANT shall be a globally static expression.
— An actual part associated with the formal parameter ResultMap shall be a locally static expression.

The concurrent signal assignment statement shall be a 0 ns delay signal assignment statement. The signal on the right hand side shall only be an output of the VITAL memory path delay section. The signal on the left hand side shall only be an output port.

 Copyright © 2001 IEEE. All rights reserved.

11. VITAL Memory function specification

The VITAL Level 1 Memory function specification supports predefined procedures and constructs for memory object declaration, memory function table specification, memory operations to perform read, write and corruption handling. The function specification also defines modeling of actions on timing constraint violations and implementing cross port propagation, contention policies in multiport memories. The various subroutines used for memory object declaration and performing functional operations are defined in Table 9.

Table 9—VITAL memory functionality procedures

Name	Description
VitalDeclareMemory	Function to declare and initialize memory object
VitalMemoryTable	Procedure to perform memory read, write and corruption
VitalMemoryCrossPorts	Procedure to implement multiport contention and cross port read
VitalMemoryViolation	Procedure to specify actions on timing constraint violation

These subroutines shall only be invoked in a VITAL Level 1 Memory process statement.

11.1 VITAL memory construction

A VITAL memory object consists of a two dimensional array of the type VitalMemoryDataType defined in VITAL_Memory package. It is the basic storage component of a VITAL memory model which is declared using a call to VitalDeclareMemory function. The memory object is organized as a set of bits which form a *memory word* and a set of memory words form which form a *memory array*.

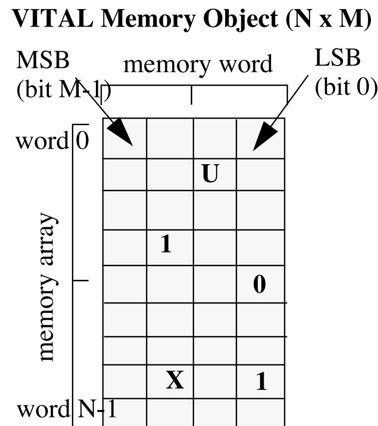


Figure 10—N x M memory array (N words and M bits per word)

Copyright © 2001 IEEE. All rights reserved.

The memory array of N x M dimension is organized from top to bottom with top-most word being addressed by index 0 and the last word being addressed by index N-1. Each memory word in this array is organized from left to right with left-most bit being most significant (MSB: M-1) and right-most bit being least significant bit (LSB: bit 0). The legal states of a bit inside the memory array shall only be one of 'U','X','0','1'. Any access to this memory object shall be consistent with the above structure of memory array.

11.1.1 Memory subword addressability

The VITAL memory specification supports the modeling of *subword addressable memories*. A set of contiguous bits which form a portion of a full memory word and the access to which is controlled by an associated enable pin is called a *memory subword*. A subword addressable memory is partitioned into a set of memory subword arrays in which each memory subword consists of equal number of bits. The only exception allowed to this rule is that the size of most significant subword can be less than the size of the other memory subwords. A memory that is accessed using all the bits of a memory word is refered to as *full word memory*.

The number of bits in each memory subword is denoted by NoOfBitsPerSubWord and is specified in a call to VitalDeclareMemory.

.

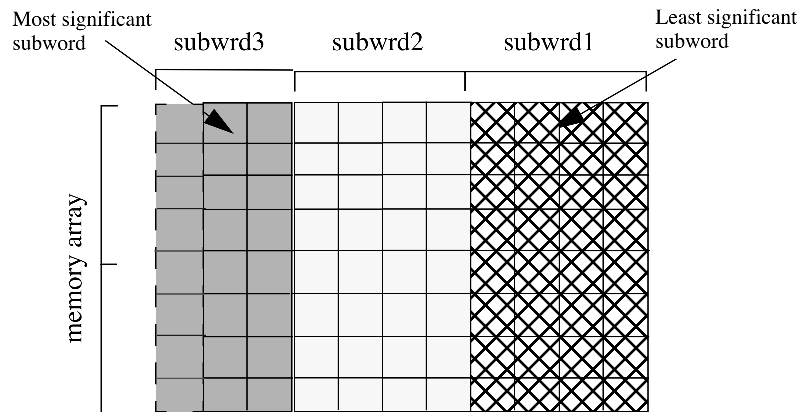


Figure 11—Memory subword addressability

NOTE— The size of a memory subword can be a single bit (bit writable memories).

11.1.2 Mutiport memory structures

The VITAL memory specification supports the modeling of memories with multiple address and data output ports. The type of address port is specified by the mode of access provided by the port. The possible mode of access can be read only, write only and read/write. In a multiport memory an association from an address port to a corresponding data output port is referred to as the *same port* and an association from an address port to any other data output port is refered to as the *cross port*. A memory access performed with this association is refered to as *same port access* and *cross port access* respectively. A memory access could be for read or write.

NOTE— ASIC memories are typically designed as addressable latches. In such cases the read only and read/write ports are always associated with a corresponding data output ports. However the write only ports do not have any direct association with data output ports.

 Copyright © 2001 IEEE. All rights reserved.

Example:

The following figure illustrates the same port and cross port structures for two different memory architectures.

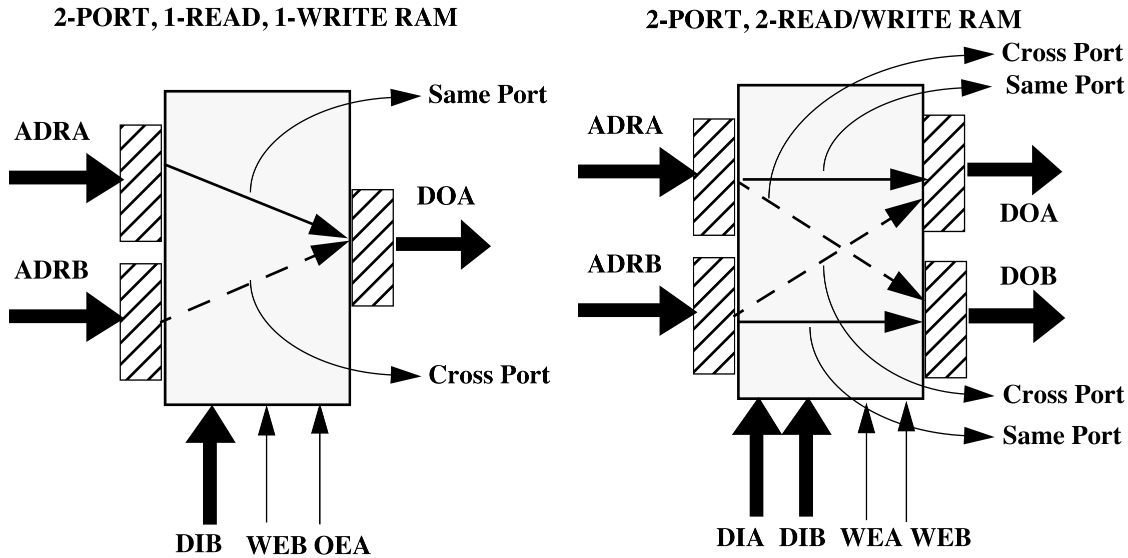


Figure 12—Memory multiport structures

A simultaneous access can occur to any memory location from different address ports in a multiport memory. This could result in a memory contention or a legal memory operation based on input conditions of the address ports. Memory contention policies and cross port propagation is implemented through a call to VitalMemoryCrossPorts. The Table 10 contains all possible memory and output actions under various input conditions for any two address ports which are defined by same port and cross port relationship. These possibilities are denoted by CrossPortMode in a call to VitalMemoryCrossPorts.

Table 10—Multiport memory contention and cross port read

Mode of same port address	Mode of cross port address	Memory action	Same port output data action	Cross port output data action
READ	READ	No change	Read memory contents	Read memory contents
READ	WRITE	Legal Write	Read memory contents with latest data written	Read memory contents with latest data written
READ	WRITE	Legal Write	Corrupt output data	Corrupt output data
READ	WRITE	Corrupt memory	Corrupt output data	Corrupt output data
WRITE	WRITE	Corrupt memory	Corrupt output data	Corrupt output data
WRITE	WRITE	Legal Write	Read memory contents with latest data written	Read memory contents with latest data written

Copyright © 2001 IEEE. All rights reserved.

:NOTES:

1) The same set of input conditions could result in different memory and data output actions depending upon memory architectures.

2) The above table assumes memories have cross port read operations as a part of their functional behavior. In synchronous ASIC memories simultaneous write is legal if two write clocks are seperated by minimum time known as access separation time.

11.2 VITAL memory table specification

The package VITAL_Memory supports standard specification and use of VITAL memory tables. A VITAL memory table is provided for modeling functional behavior of the memory model. The table is implemented through a lookup mechanism which determines the memory and output data action to perform based on inputs to and other conditions in the memory model. These tables can be of two different forms viz., *memory function table* and *memory violation table*. A memory function table (represented by MemoryTable parameter) is used to represent functional operations such as read, write and unknown input conditions and is accessed using VitalMemoryTable procedure calls in a Level 1 memory model. A memory violation table (represented by ViolationTable parameter) is used to model actions based on timing constraint violations and is accessed using a call to VitalMemoryViolation procedure in a Level 1 memory model. The structure of memory tables is similar to VITAL state table specification (see 7.3).

11.2.1 Memory table partitioning

A VITAL memory table is partitioned into different sections which represent a specific kind of information. The two major categories are the *input section*, and the *response section*. The input section can be further classified into the *direct input columns* and *interpreted input columns*. The response section consists of the *memory action column* and the *output action column*. The semantics of interpreted input column specification and processing differs between VitalMemoryTable and VitalMemoryViolation procedures. The definition of various input and response sections for MemoryTable and ViolationTable are as follows:

— The sequence of columns in a MemoryTable (ViolationTable) shall only be direct inputs, followed by interpreted inputs, followed by memory action and output action columns.
— In the case of MemoryTable, the direct input columns are always associated with scalar inputs and they represent the input pattern applied to the memory on those inputs.
— In the case of ViolationTable, the direct input columns are always associated with scalar violation flags and they represent the current state of these flags.
— The interpreted input columns are always associated with vector inputs. The table literal specified in the interpreted input column represent the kind of memory table symbol semantics to be matched with corresponding state of vector inputs.
— The MemoryTable specification consists of three possible kinds of interpreted input columns which correspond to EnableBus, AddressBus and DataInBus. The order of these columns is significant and the MemoryTable specification shall adhere to this sequence. The EnableBus column is only required in the case of subword addressable memories.
— The ViolationTable consists of as many interpreted input columns as the number of vector violation flags specified as part of ViolationFlagsArray in a call to VitalMemoryViolation.
— The memory action column determines the operation on the contents of the memory based on direct and interpreted input columns.
— The output action column determines the value returned on the memory output databus based on direct and interpreted input columns.

 Copyright © 2001 IEEE. All rights reserved.

11.2.2 VITAL memory table symbols

A transition set or a steady state condition in a memory model is represented by a special memory table symbol. The symbol set defined by the type VitalMemoryTableSymbolType is used to specify memory tables.

```
type VitalMemorySymbolType is (
    '/',    -- 0 -> 1
    '\',    -- 1 -> 0
    'P',    -- Union of '/' and '^' (any edge to 1)
    'N',    -- Union of '\' and 'v' (any edge to 0)
    'r',    -- 0 -> X
    'f',    -- 1 -> X
    'p',    -- Union of '/' and 'r' (any edge from 0)
    'n',    -- Union of '\' and 'f' (any edge from 1)
    'R',    -- Union of '^' and 'p' (any possible rising edge)
    'F',    -- Union of 'v' and 'n' (any possible falling edge)
    '^',    -- X -> 1
    'v',    -- X -> 0
    'E',    -- Union of 'v' and '^' (any edge from X) or Corrupt a subword with 'X' based on data
in
    'A',    -- Union of 'r' and '^' (rising edge to or from 'X')
    'D',    -- Union of 'f' and 'v' (falling edge to or from 'X') or  Corrupt a single bit with 'X' in
subword
    '*',    -- Union of 'R' and 'F' (any edge)
    'X',    -- Unknown level
    '0',    -- low level
    '1',    -- high level
    '-',    -- don't care
    'B',    -- 0 or 1
    'Z',    -- High Impedance
    'S'     -- steady value
    'g',    -- Good address (no transition)
    'u',    -- Unknown address (no transition)
    'i',    -- Invalid address (no transition)
    'G',    -- Good address (with transition)
    'U',    -- Unknown address (with transition)
    'I',    -- Invalid address (with transition)
    'w',    -- Write data to memory
    's',    -- Retain previous memory contents
    'c',    -- Corrupt entire memory with 'X'
    'l',    -- Corrupt a word in memory with 'X'
    'd',    -- Corrupt a single bit in memory with 'X'
    'e',    -- Corrupt a word with 'X' based on data in
    'C',    -- Corrupt a subword entire memory with 'X'
    'L',    -- Corrupt a subword in memory with 'X'
    'M',    -- Implicit read data from memory
    'm',    -- Read data from memory
    't'     -- Immediate assign/transfer data in
);
```

Copyright © 2001 IEEE. All rights reserved.

11.2.2.1 Memory table symbol scoping rules

The acceptable range of memory table symbols and its applicability to different sections of the table, depends on the type of table specification (MemoryTable or ViolationTable) and the association between the column position of the symbol in the table and the corresponding input parameters.

The legal range of symbols for MemoryTable specification is defined in Table 11:

Table 11—Legal symbols and scope for Memory Function Table

Section of Table	Legal Symbols
Direct Input Columns	‘/’, ‘\’, ‘P’, ‘N’, ‘r’, ‘f’’, ‘p’, ‘n’, ‘R’, ‘F’, ‘^’, ’v’, ‘E’, ‘A’, ‘D’, ’*’, ’X’, ’0’, ’1’, ’-’, ’B’, ’S’
EnableBus Interpreted Input Column	‘/’, ‘\’, ‘P’, ‘N’, ‘r’, ‘f’’, ‘p’, ‘n’, ‘R’, ‘F’, ‘^’, ’v’, ‘E’, ‘A’, ‘D’, ’*’, ’X’, ’0’, ’1’, ’-’, ’B’, ’S’
AddressBus Interpreted Input Column	‘g’, ‘u’, ‘i’, ‘G’, ‘U’, ‘I’, ‘*’, ‘-’, ‘S’
DataInBus Interpreted Input Column	‘g’, ‘u’, ‘G’, ‘U’, ‘*’, ‘-’, ‘S’
Memory Action Column	‘0’, ‘1’, ‘w’, ‘s’, ‘c’, ‘l’, ‘d’, ‘e’, ‘C’, ‘D’, ‘E’, ‘L’
Output Action Column	‘0’, ‘1’, ‘Z’, ‘l’, ‘d’, ‘e’, ‘C’, ‘D’, ‘E’, ‘L’, ‘M’, ‘m’, ‘t’, ‘S’

The legal range of symbols for ViolationTable specification is defined in Table 12:

Table 12—Legal symbols and scope for Memory Violation Table

Section of Table	Legal Symbols
Direct Input Columns	‘0’, ‘X’, ‘-’, ‘S’
Interpreted Input Columns	‘0’, ‘X’, ‘-’, ‘S’
Memory Action Column	‘0’, ‘1’, ‘s’, ‘c’, ‘l’, ‘d’, ‘e’, ‘C’, ‘D’, ‘E’, ‘L’
Output Action Column	‘0’, ‘1’, ‘l’, ‘d’, ‘e’, ‘D’, ‘E’, ‘L’, ‘S’

In the case of memory violation table, the VitalMemoryTableType symbols of ‘X’, ‘0’, ‘-’ correspond to “violation occurred”, “no violation occurred” and “don’t care” states.

The following table shows the VitalMemorySymbolType elements for the direct input columns and the level and edge transitions that they represent.

 Copyright © 2001 IEEE. All rights reserved.

Table 13—Memory table symbol semantics for direct/EnableBus interpreted input columns

	00	10	X0	11	01	X1	0X	1X	XX
‘/‘					*				
‘\‘		*							
‘P’					*	*			
‘N’		*	*						
‘r’							*		
‘f’								*	
‘p’					*		*		
‘n’		*						*	
‘R’					*	*	*		
‘F’		*	*					*	
‘^’						*			
‘v’			*						
‘E’			*			*			
‘A’						*	*		
‘D’			*					*	
‘*’		*	*		*	*	*	*	
‘X’							*	*	*
‘0’	*	*	*						
‘1’				*	*	*			
‘-’	*	*	*	*	*	*	*	*	*
‘B’	*	*	*	*	*	*			
‘Z’									
‘S’	*			*					

The following table shows the VitalMemorySymbolType symbols that have unique interpretation to memory modeling and used as part of memory function table specification. The table symbols are case specified.

Copyright © 2001 IEEE. All rights reserved.

Table 14—Memory function table symbol semantics

Symbol	Description
'*'	Any edge on any bit
'0'	For memory action and output response columns write low level on all bits. For EnableBus column mark one or more bits with low level.
'1'	For memory action and output response columns write high level on all bits. For EnableBus column mark one or more bits with high level.
'-'	Don't care (always match)
'Z'	Write or output high impedance on all bits
'S'	Steady value (no edges on any bit)
'g'	Valid (good) address or data inputs
'u'	Unknown values ('X') on address or data inputs
'i'	Invalid inputs on address (Address out of range)
'G'	Any change and valid (good) address or data inputs
'U'	Any Change and Unknown inputs 'X'
'I'	Any Change & Invalid inputs (out of range)
'w'	Write data to memory
's'	Retain previous memory contents
'c'	Corrupt entire memory with 'X'
'l'	Corrupt a word in memory with 'X'
'd'	Corrupt a single bit in memory with 'X'
'e'	Corrupt a word with 'X' based on data in
'D'	For memory action and output response columns, corrupt a single bit in memory with 'X' in subword. For EnableBus coulmn see Table 4.
'E'	For memory action and output response columns, corrupt a memory subword with 'X' based on data in. For EnableBus coulmn see Table 4.
'C'	Corrupt a memory subword entire memory with 'X'
'L'	Corrupt a memory subword in memory with 'X'
'M'	Implicit read data from memory
'm'	Read data from memory
't'	Immediate transfer data in (transparent mode)

 Copyright © 2001 IEEE. All rights reserved.

11.2.3 VITAL memory table symbol matching

During memory table processing, the inputs to the VitalMemoryTable (VitalMemoryViolation) procedure are matched to the stimulus portion of the MemoryTable (ViolationTable). The inputs consists of direct inputs and interpreted inputs. The matching scheme used differs between VitalMemoryTable and VitalMemoryViolation.

11.2.3.1 Memory function table symbol matching

The symbol matching scheme used in memory function table processing depends upon on the section of the MemoryTable being matched:

— For the direct input columns, the matching process begins by converting the input data (Controls) to the equivalent 'X', '0', or '1' values by applying the standard logic TO_X01 function. The resulting values are then compared to the stimulus portion of the table according to the following rules:

Table 15—Matching of table symbols to input stimulus

Table stimulus portion	ResultX01 := To_X01 (Direct Inputs)	Result of comparison
'X'	'X'	'X' only matches with 'X'
'0'	'0'	'0' only matches with '0'
'1'	'1'	'1' only matches with '1'
'-'	'X', '0', '1'	'-' matches with any value of input X01
'B'	'0', '1'	'B' only matches with '0' or '1'

The current and previous values of direct inputs are used to determine if an edge has occurred. These edges are matched with the edge entries which are specified in the input pattern of the table using the semantics of the edge symbols shown in Table 4.

— The interpreted input columns corresponding to AddressBus or DataInBus are matched against the address or data inputs respectively, in a two stage process:

a) The values corresponding to AddressBus or DataInBus are converted into interpreted input literal internally as defined in Table 16:

Table 16—Interpreted input conversion for AddressBus and DataInBus

Interpretation	Symbol
Steady state condition on all bits of AddressBus or DataInBus. No 'X' present.	'g'
Steady state condition on all bits of AddressBus or DataInBus. One or more bits in 'X' state.	'u'
Steady state condition and invalid state due to address out of range condition on AddressBus. No 'X' present.	'i'

Copyright © 2001 IEEE. All rights reserved.

Table 16—Interpreted input conversion for AddressBus and DataInBus (continued)

Interpretation	Symbol
Transition on one or more bits of AddressBus or DataInBus. No 'X' present.	'G'
Transition on one or more bits of AddressBus or DataInBus. One or more bits in 'X' state.	'U'
Transition on or more bits and invalid state due to address out of range condition on AddressBus. No 'X' present.	'I'

b) The interpreted input literal derived in the previous step is matched against the symbol present in the corresponding interpreted input column of the memory function table as defined Table 17:

Table 17—Interpreted input column matching forAddressBus and DataInBus \ (memory table symbols in rows, interpreted input literal in columns)

	'g'	'u'	'i'	'G'	'U'	'I'
'g'	*					
'u'		*				
'i'			*			
'G'				*		
'U'					*	
'I'						*
'*'				*	*	*
'-'	*	*	*	*	*	*
'S'	*	*	*			

NOTE— The interpreted input columns converted to a To-X01 values.

— The matching process for the interpreted input columns corresponding to EnableBus is accomplished on a bit-by-bit basis. The value of each bit is matched against the symbol present in the corresponding interpreted input column of the memory function table similar to the matching process used for direct input columns. If edge entries are specified for a symbol, the current and previous values of EnableBus are used to determine if edge has occured.

11.2.3.2 Memory violation table symbol matching

The symbol matching scheme used in memory violation table processing depends upon on the section of the ViolationTable being matched.

— For the direct input columns, the matching process is done by comparing input data (ViolationFlags) to the stimulus portion of the table according to the following rules:

 Copyright © 2001 IEEE. All rights reserved.

Table 18—Matching of table symbols to input stimulus

Table stimulus portion	**ResultX01 :=** (Direct Inputs)	**Result of comparison**
'X'	'X'	'X' only matches with 'X'
'0'	'0'	'0' only matches with '0'
'-'	'X', '0', '1'	'-' matches with any value of input 'X','0','1'

— The matching process for the interpreted input columns corresponding to ViolationFlagsArray is accomplished on a bit-by-bit basis. The value of each bit is matched against the symbol present in the corresponding interpreted input column of the memory violation table similar to the matching process used for direct input columns.

11.2.4 VITAL memory table construction

A VITAL memory table forms the basis of modeling a Level 1 compliant memory model. The memory tables can be of two different forms: memory function table and memory violation table.

A memory table is defined as a two dimensional object of type VitalMemoryTableType.

```
type VitalMemoryTableType is array (Natural range <>, Natural range <>)
  of VitalMemorySymbolType;
```

The length of the first dimension (number of rows) of a memory table shall be the number of input conditions that have a specified memory action and output response. The length of the second dimension (number of columns) shall be the sum of the length of the input pattern section (including direct inputs and interpreted inputs), the memory action and output response section.

11.2.4.1 Memory function table construction

A memory function table, represented by MemoryTable, is used to model functional operations of the memory such as reading, writing and taking actions on unknown input conditions. VitalMemoryTable procedure calls are used to access MemoryTable in a Level 1 memory model.

The memory function table consists of input section (direct and interpreted), memory action and output action section (see 11.2). The Controls, EnableBus, AddressBus, and DataInBus parameters are used to form the input portion of the MemoryTable which is passed to VitalMemoryTable procedure. If multiple enable inputs need to be specified, the actual parameters corresponding to each input shall be passed as a concatenation to the EnableBus input parameter. Each vector enable input to the EnableBus parameter shall be of the same size. The presence of EnableBus parameter is optional.

The number of inputs to the memory function table shall be equal to sum of length of Controls, EnableBus (if specified), AddressBus, and DataInBus. It is an error if the length of inputs is greater or equal to the size of the second dimension of MemoryTable parameter.

The symbol literals specified in the direct input columns correspond to the Controls parameter of VitalMemoryTable from left to right in a one-to-one relationship.

Copyright © 2001 IEEE. All rights reserved.

The symbol literals specified in the interpreted input columns correspond to EnableBus, AddressBus and DataInBus parameters of VitalMemoryTable from left to right in a one-to-one relationship. If multiple enable inputs are specified in EnableBus, each such input must correspond to a column in the interpreted input columns from left to right.

Example:

```
----------------------------------------------------
-- Memory Function Table
----------------------------------------------------
CONSTANT MemoryTable_VM2P1R1WLS : VitalMemoryTableType := (

  --  ----------------------------------------------------------
  --  AddrF, EN,  CLK, WEB, ADR, DIB, act, DOA;
  --  ----------------------------------------------------------

    ( '0',   '1', '/', '-', 'g', '-', 's', 'm' ), -- Read
    ( '0',   '1', '/', '-', 'u', '-', 's', 'l' ),
    ( '0',   '1', '/', '-', 'i', '-', 's', 'l' ),

    ( '1',   '1', '/', '1', 'g', '-', 'w', 'S' ), -- Write
    ( '1',   '1', '/', '1', 'u', '-', 'c', 'l' ),
    ( '1',   '1', '/', '1', 'i', '-', 'c', 'l' ),
    ( '1',   '1', '/', '0', '-', '-', 's', 'S' ),

    ( '-',   '0', '-', '-', '-', '-', 's', 'S' ), -- EN low

    ( '0',   '*', '1', '-', '-', '-', 's', 'l' ), -- EN changes
    ( '1',   '*', '1', '-', '-', '-', 'c', 'l' ),

    ( '0',   'X', '/', '-', '-', '-', 's', 'l' ), -- EN 'X'
    ( '1',   'X', '/', '-', '-', '-', 'c', 'l' ),

    ( '1',   '1', '/', 'X', '-', '-', 'e', 'S' )  -- WEB 'X'

);
```

11.2.4.2 Memory violation table

A memory violation table, represented by ViolationTable parameter, is used to model actions based on timing constraint violations. The VitalMemoryViolation procedure call is used to access ViolationTable in a Level 1 memory model.

The memory violation table consists of input section (direct and interpreted), memory action and output action section (see 11.2). The input portion of the ViolationTable is formed using the ViolationFlags, and ViolationFlagsArray parameter passed to VitalMemoryViolation procedure. If multiple vector violation flags need to be specified, the actual parameters corresponding to each input shall be passed as a concatenation to the ViolationFlagsArray input parameter. The presence of ViolationFlagsArray parameter is optional.

The number of inputs to the memory function table shall be equal to sum of length of ViolationFlags and ViolationFlagsArray. It is an error if the length of inputs is greater or equal to the size of the second dimension of ViolationTable parameter.

 Copyright © 2001 IEEE. All rights reserved.

The symbol literals specified in the direct input columns correspond to the ViolationFlags parameter of VitalMemoryViolation from left to right in a one-to-one relationship.

The symbol literals specified in the interpreted input columns correspond to ViolationFlagsArray parameter of VitalMemoryViolation from left to right in a one-to-one relationship. If multiple sets of vector violation flags are specified in ViolationFlagsArray, each such input must correspond to a column in the interpreted input columns from left to right. The size of each such set of violation flags is specified as a concatenation to the ViolationSizesArray parameter in a call to VitalMemoryViolation.

Example:

```
        -- Memory Violation Table
        CONSTANT  ViolTable_VM3P1R1WLA : VitalMemoryTableType := (
         -- V_CADR_R  V_CADR_F  V_WEC_R  V_DIC: V_AADR: V_BADR act : DOA/
B
           ( 'X',     '-',      '-',     '-',   '-',    '-',   'c',  'l'
),
           ( '-',     'X',      '-',     '-',   '-',    '-',   'c',  'l'
),
           ( '-',     '-',      'X',     '-',   '-',    '-',   'c',  'l'
),
           ( '-',     '-',      '-',     'X',   '-',    '-',   'd',  'l'
),
           ( '-',     '-',      '-',     '-',   'X',    '-',   's',  'l'
),
           ( '-',     '-',      '-',     '-',   '-',    'X',   's',  'l'
)
         );
```

11.3 VitalDeclareMemory

A VITAL memory object (see 11.1) is represented by a variable of type VitalMemoryDataType and is declared using a call to VitalDeclareMemory function. This memory variable shall be passed as an actual to the formal parameter MemoryData in a call to VitalMemoryTable, VitalMemoryCrossPorts and VitalMemoryViolation.

The VitalDeclareMemory function has two different overloadings. One overloading for subword memory devices and the other overloading is for full word memory devices. VitalDeclareMemory can also be used to initialize the contents of the VITAL memory object from a memory load file. This feature can be used for memory pre-load operations in static ROMs and RAMs. The following table explains the formal arguments of the VitalDeclareMemory function :

Table 19—Formal arguments for VitalDeclareMemory

Argument Name	Argument Type	Argument Description
NoOfWords	POSITIVE	Number of words in the memory array
NoOfBitsPerWord	POSITIVE	Number of bits in a memory word
NoOfBitsPerSubWord	POSITIVE	Number of bits in one memory subword
MemoryLoadFile	STRING	Path to the memory load file with which the memory contents can be initialized
BinaryLoadFlag	BOOLEAN	Flag to indicate binary load format. Default value is FALSE indicating that the memory data is in hexadecimal format.

Copyright © 2001 IEEE. All rights reserved.

NOTE— A single memory object can be declared and passed between multiple calls to the VitalMemoryTable and other memory modeling procedures. This is used to model the shared memory and cross port access characteristics of multi-port memories.

11.3.1 Memory Initialization

The VitalDeclareMemory function can be used to initialize a memory object of type VitalMemoryDataType from a memory pre-load file. The contents of the memory pre-load file can either be in hexadecimal format or binary format denoted by BinaryLoadFlag parameter. The default value of this flag is FALSE indicating hexadecimal format of memory data file. The syntax of the pre-load file is defined as follows:

VITAL_memory_load_file ::=

{ VITAL_memory_load_data_statement }

VITAL_memory_load_data_statement ::=

[VITAL_memory_addr_specification] <memory_data_value>

VITAL_memory_addr_specification ::= @ address value in_hexadecimal

memory_data_value ::= data word in binary or hexadecimal

The memory pre-load file can contain multiple VITAL memory load data statements. The address value shall only be in hexadecimal format. The data value shall be in either hexadecimal or binary format. VitalDeclareMemory will interpret the organization of the data word consistent with the organization of the memory word (see 11.1). VitalDeclareMemory reads each memory load data statement from the pre-load file and assigns the data word to the memory word indicated by the address value, starting from right (LSB) to left (MSB). The presence of address value in the pre-load file is optional. If the address value is not specified, VitalDeclareMemory shall assign the data word in ascending order of address starting with first data word being assigned to the memory word corresponding to index address of the memory array. Each data word shall be assigned to a successive memory word in the memory array in this case.

If the size of the data word is less than the size of the memory word, VitalDeclareMemory will only fill a portion of the memory word equal to the number of bits in the data value starting from right (LSB) to left (MSB).

If the size of data word is more than the size of the memory word, VitalDeclareMemory will assign as many bits from the data word as defined by the size of the memory word starting from right (LSB) to left (MSB).

All bits in the data word are converted to equivalent ‘X’, ‘0’, or ‘1’ values by applying the standard logic TO_X01 function before assignment.

Example :

Memory Load File with address information and hexadecimal data:

```
@0  aa
@1  bb
@2  cc
@3  dd
@a  ee
@b  ff
```

 Copyright © 2001 IEEE. All rights reserved.

Memory Load file without address information, binary data:

```
110101
110111
111011
101111
```

11.4 VitalMemoryTable

The VITAL Level 1 memory function specification support the modeling memory function tables using the predefined procedure VitalMemoryTable. It is used to perform basic memory operations such as read, write and actions on unknown input conditions.

There shall be one VitalMemoryTable call per address port in a Level 1 memory model all of which are contained within a single process. A single memory function table may be shared by more than one VitalMemoryTable procedures. The VitalMemoryTable procedure support the modeling of subword addressable memories by providing two overloaded versions of this procedure; one works with full word memories and the other with subword memories.

The following table explains the formal arguments of the VitalMemoryTable function:

Table 20—Formal arguments for VitalMemoryTable

Argument Name	Argument Type	Argument Description
DataOutBus	STD_LOGIC_VECTOR	Variable for Functional Output Data
MemoryData	VitalMemoryDataType	Pointer to VITAL memory data object
PrevControls	STD_LOGIC_VECTOR	Previous state values of Controls parameter
PrevEnableBus	STD_LOGIC_VECTOR	Previous state values of PrevEnableBus parameter
PrevDataInBus	STD_LOGIC_VECTOR	Previous DataInBus for edge detection
PrevAddressBus	STD_LOGIC_VECTOR	Previous address bus for edge detection
PortFlag	VitalPortFlagType	Indicates operating mode of the port in a single process execution. The possible values are READ, WRITE, CORRUPT, NOCHANGE.
PortFlagArray	VitalPortFlagVectorType	Vector form of PortFlag for subword addressable memories. Used in overloaded version.
Controls	STD_LOGIC_VECTOR	Aggregate of scalar memory control inputs
EnableBus	STD_LOGIC_VECTOR	Concatenation of vector control inputs
DataInBus	STD_LOGIC_VECTOR	Memory data in bus inputs
AddressBus	STD_LOGIC_VECTOR	Memory address bus inputs
AddressValue	VitalAddressValueType	Decoded integer value of the AddressBus
MemoryTable	VitalMemoryTableType	Memory function table

Copyright © 2001 IEEE. All rights reserved.

Table 20—Formal arguments for VitalMemoryTable (continued)

Argument Name	Argument Type	Argument Description
PortType	VitalPortType	The base type of port (one of READ, WRITE, RDNWR)
PortName	STRING	Port name string for messages
HeaderMsg	STRING	Header string for messages
MsgOn	BOOLEAN	Reporting control of message generation
MsgSeverity	SEVERITY_LEVEL	Severity control of message generation

Example:

```
VitalMemoryTable (
        DataOutBus          =>      DOA_zd,
        MemoryData          =>      MemoryData_VM2P1R1WLS,
        PrevControls        =>      PrevControls_A,
        PrevEnableBus       =>      PrevEnableBus_A,
        PrevDataInBus       =>      PrevDataInBus_A,
        PrevAddressBus      =>      PrevAddressBus_A,
        PortFlagArray       =>      PortFlag_A,
        Controls            =>      (ReadAddrFlag,ENA_i,CLKA_i),
        EnableBus           =>      WEB_i_dly,
        DataInBus           =>      DIB_i_dly,
        AddressBus          =>      AADR_i_dly,
        AddressValue        =>      AdrValue_A,
        MemoryTable         =>      MemoryTable_VM2P1R1WLS,
        PortName            =>      "Port A",
        HeaderMsg           =>      "READ ONLY"
);
```

11.4.1 VitalMemoryTable algorithm

The VitalMemoryTable procedure compute the memory action and output response based on various inputs and the present state. These procedures perform a row dominant sequential search on the MemoryTable. It compares the stimulus with the input pattern section of each row in the memory table. The comparison of each entry continues until all of the inputs have compared or a mismatch is encountered. The search terminates when a matching entry for all inputs is found or when the table entries are exhausted. If the memory function table has been searched and no match is found, no action is taken. If a matching row is found, the output response and memory action is determined from the corresponding columns of that row. The outputs are then converted to standard logic X01Z subtype and propagated to DataOutBus parameter. Similarly memory array object pointed to by MemoryData is updated with the corresponding memory action determined as a result of table evaluation. At the end of each memory processs execution the PortFlag parameter captures the possible states (as defined by READ, WRITE, CORRUPT and NOCHANGE) indicating the operating mode of the corresponding memory address port. The PortFlag is passed to VitalMemoryViolation and VitalMemoryCrossPorts procedures to convey the result of memory function table evaluation.

 Copyright © 2001 IEEE. All rights reserved.

11.5 VitalMemoryCrossPorts

The VITAL Level 1 memory function specification supports the modeling of memory contention policies and cross port data propagation in multiport memories, using the predefined procedure VitalMemoryCrossPorts. It is used to model cross port read operations from the same memory location which is being written to using the corresponding write address port. It is also used to model memory contention behavior between multiple write ports and in some cases contention due to read from the same memory location for which a write cycle is in progress. Table 10 captures the various cross port read and memory contention situations supported by VITAL Level 1 memory function specification.

In multiport memory models, there shall be one VitalMemoryCrossPorts invocation per data output port in a Level 1 memory model. If there are multiple write only or read/write ports an additional invocation is needed to model the effect of memory contention among such write ports. There are two overloaded versions of VitalMemoryCrossPorts procedure calls defined in the package corresponding to the above mentioned situation.

```
PROCEDURE VitalMemoryCrossPorts (
    VARIABLE MemoryData                : INOUT VitalMemoryDataType;
    CONSTANT CrossPortFlagArray        : IN VitalPortFlagVectorType;
    CONSTANT CrossPortAddressArray     : IN VitalAddressValueVectorType;
    CONSTANT HeaderMsg                 : IN STRING := "";
    CONSTANT MsgOn                     : IN BOOLEAN := TRUE
) ;

PROCEDURE VitalMemoryCrossPorts (
    VARIABLE DataOutBus                : INOUT std_logic_vector;
    VARIABLE MemoryData                : INOUT VitalMemoryDataType;
    CONSTANT SamePortFlag              : IN VitalPortFlagVectorType;
    CONSTANT SamePortAddressValue      : IN VitalAddressValueType;
    CONSTANT CrossPortFlagArray        : IN VitalPortFlagVectorType;
    CONSTANT CrossPortAddressArray     : IN VitalAddressValueVectorType;
    CONSTANT CrossPortMode             : IN VitalCrossPortModeType
                                           := CpReadAndWriteContention;
    CONSTANT PortName                  : IN STRING := "";
    CONSTANT HeaderMsg                 : IN STRING := "";
    CONSTANT MsgOn                     : IN BOOLEAN := TRUE
) ;
```

The first overloading of VitalMemoryCrossPorts is intended to model cross port write contention between multiple write-only and/or multiple read-write ports. This procedure shall only be used if there are more than one such port capable of writing to the same memory location during a single memory cycle. The second overloading of VitalMemoryCrossPorts is intended to model cross port read operations and memory contention situations between multiple address ports as defined by the values of CrossPortMode parameter. Table 21 captures the various operating modes of VitalMemoryCrossPorts based on the actual part associated with CrossPortMode parameter.

Copyright © 2001 IEEE. All rights reserved.

Table 21—Operating modes for VitalMemoryCrossPorts

CrossPortMode Enumerations	CrossPortMode Description
CpRead	Only cross port reads are performed with the new datain being written. No memory contention checks.
WriteContention	Only write contention checks are performed among multiple ports capable of writing. No cross port reads occur.
ReadWriteContention	Contention occurs between ports accessing same memory location for reading and writing. Both memory and data outputs are corrupted.
CpReadAndReadContention	Contention occurs between ports accessing same memory location for reading and writing. Data outputs are corrupted but memory contents are not affected.
CpReadAndWriteContention	Crossport reads occur and write contention checks are performed only among multiple ports capable of writing.

Table 22 explains the formal arguments of the VitalMemoryCrossPorts procedure:

Table 22—Formal arguments for VitalMemoryCrossPorts

Argument Name	Argument Type	Argument Description
DataOutBus	STD_LOGIC_VECTOR	Variable for Functional Output Data
MemoryData	VitalMemoryDataType	Pointer to VITAL memory data object
SamePortFlag	VitalPortFlagVectorType	Indicates operating mode of the same port in a single process execution.
SamePortAddressValue	VitalAddressValueType	Decoded integer value of the same port address.
CrossPortFlagArray	VitalPortFlagVectorType	Indicates operating mode of the various cross ports in a single process execution.
CrossPortAddressArray	VitalAddressValueVectorType	An array of decoded integer value of various cross port addresses.
CrossPortMode	VitalCrossPortModeType	Enumeration literal to select functional mode of VitalMemoryCrossPorts.
PortName	STRING	Name of the cross port being checked.
HeaderMsg	STRING	Header string for messages
MsgOn	BOOLEAN	Reporting control of message generation

Copyright © 2001 IEEE. All rights reserved.

Example:

```
-- Cross Ports

VitalMemoryCrossPorts (
   DataOutBus              => DOA_zd,
   MemoryData              => MemoryData_VM3P2R1WLA,
   SamePortFlag            => PortFlag_A,
   SamePortAddressValue    => AdrValue_A,
   CrossPortFlagArray      => PortFlag_C,
   CrossPortAddressArray   => (0 => AdrValue_C),
   PortName                => "Port A"
);
```

11.5.1 VitalMemoryCrossPorts functionality

The VitalMemoryCrossPorts procedure perform various functions related to cross port data propagation and memory contention checking during simultaneous access to single memory location. The operation performed is based on the CrossPortMode chosen. This procedure is specified for each data output port on which a cross port read operation is desired. The address port corresponding to this data output port is established as the reference port for address comparisons and is refered to as the *same port*. All other address ports capable of cross port access to this data output port are refered to as *cross ports*.

During simultaneous access to any memory location, a cross port data propagation is initiated, if the value of the SamePortFlag is set to READ from a preceeding VitalMemoryTable procedure call. VitalMemoryCrossPorts will determine the candidate port from a set of cross ports that is currently in a WRITE mode. The address of the reference port (SamePortAddressValue) is compared with the address of the identified candidate cross port. If a match is found, then the value of the data output port (DataOutBus) associated with the reference port is updated with the contents of addressed memory location. This allows the data output port to be updated with latest data as a result of WRITE cycle performed by cross port. A memory contention is detected if multiple ports including the reference port are in WRITE mode. The memory and data output action in these cases is based on the CrossPortMode selected and is defined in Table 10 and Table 21.

11.6 VitalMemoryViolation

The VITAL Level 1 memory function specification supports the modeling of memory violation tables using the predefined procedure VitalMemoryViolation. It is used to model specific actions on timing constraint violations. The various memory corruption policies required to invalidate (assign 'X') memory contents and data outputs can be specified through memory violation table. The VitalMemoryViolation procedure support subword addressable memories by providing two overloaded versions; one works with full word memories and the other with subword memories.

Table 23 explains the formal arguments of the VitalMemoryViolation procedure:

Copyright © 2001 IEEE. All rights reserved.

Table 23—Formal arguments for VitalMemoryViolation

Argument Name	Argument Type	Argument Description
DataOutBus	STD_LOGIC_VECTOR	Variable for Functional Output Data
MemoryData	VitalMemoryDataType	Pointer to VITAL memory data object
PortFlag	VitalPortFlagType	Indicates operating mode of the port in a single process execution. The possible values are READ, WRITE, CORRUPT, NOCHANGE.
DataInBus	STD_LOGIC_VECTOR	Memory data in bus inputs
AddressValue	VitalAddressValueType	Decoded integer value of the AddressBus
ViolationFlags	STD_LOGIC_VECTOR	Contains a list of scalar violation variables set in timing checks.
ViolationFlagsArray	X01ArrayT	Contains a list of vector violation variables set in timing checks.
ViolationSizesArray	VitalMemoryViolFlagSizeType	An integer array containing the size of each element in ViolationFlagsArray.
ViolationTable	VitalMemoryTableType	Memory violation table used to specify actions on timing violations.
PortType	VitalPortType	The base type of port (one of READ, WRITE, RDNWR)
PortName	STRING	Port name string for messages
HeaderMsg	STRING	Header string for messages
MsgOn	BOOLEAN	Reporting control of message generation
MsgSeverity	SEVERITY_LEVEL	Severity control of message generation

Example:

```
VitalMemoryViolation (
   DataOutBus            => DOA_zd,
   MemoryData            => MemoryData_VM3P2R1WLA,
   PortFlag              => PortFlag_A,
   DataInBus             => DIC_i,
   AddressValue          => AdrValue_C,
   ViolationFlags        => ( 0 => Viol_CADR_WECR,
                              1 => Viol_CADR_WECF,
                              2 => Viol_WECR ),
   ViolationFlagsArray   => ( Viol_BADR_RF & Viol_AADR_RF & Viol_DIC_WEC ),
   ViolationSizesArray   => ( 0 => Viol_DIC_WEC'LENGTH,
                              1 => Viol_AADR_RF'LENGTH,
                              2 => Viol_BADR_RF'LENGTH ),
```

 Copyright © 2001 IEEE. All rights reserved.

```
        ViolationTable           => ViolTable_VM3P1R1WLA,
        PortType                 => WRITE,
        PortName                 => "Port C",
        HeaderMsg                => "*",
        MsgOn                    => MsgOn,
        MsgSeverity              => MsgSeverity
    );
```

11.6.1 VitalMemoryViolation algorithm

The VitalMemoryViolation procedure compute the memory action and output response based on state of timing violation flags set by VITAL timing check procedures and passed to ViolationFlags and ViolationFlagsArray parameter. These procedures perform a row dominant sequential search on the ViolationTable. It compares the input stimulus (indicates the state of violation variables set during timing violations) with the input pattern section of each row in the memory violation table. The comparison of each entry continues until all of the inputs have compared or a mismatch is encountered. The search terminates when a matching entry for all inputs is found or when the table entries are exhausted. If the memory violation table has been searched and no match is found, no action is taken. If a matching row is found, the output response and memory action is determined from the corresponding columns of that row. The outputs are then converted to standard logic X01Z subtype.

Copyright © 2001 IEEE. All rights reserved.

12. VITAL memory timing specification

The VITAL Level 1 Memory timing specification supports predefined procedures for propagation delay path selection, signal output scheduling and timing constraint checking in a VITAL Level 1 Memory model. The procedures used for delay propagation are VitalMemoryInitPathDelay, VitalMemoryAddPathDelay and VitalMemorySchedulePathDelay and shall only be used in a Level 1 Memory process statement. The memory path delay procedures choose the appropriate propagation delay path, select the transition dependent delay and schedule the functional output value on the specified output signal. The procedures used for timing checks in memories are VitalMemorySetupHoldCheck and VitalMemoryPeriodPulseCheck. These procedures perform timing constraint checks and upon detection of a timing constraint violation, they report the violation and set violation variable(s) to 'X'.

12.1 VITAL memory timing types

The VITAL Level 1 Memory timing specification uses timing types and subtypes declared in the VITAL_Timing package and in the VITAL_Memory package.

12.1.1 VITAL memory delay types and sub types

The following delay types and subtypes declared in the VITAL_Timing package are used for specification and selection of simple, transition dependent path delay values (see 9.1) and output retain delay values (see 12.3). The following is the list of supported delay types and sub types used in VITAL Level 1 memory timing specification:

VitalDelayType
VitalDelayType01
VitalDelayType01Z
VitalDelayType01ZX
VitalDelayArrayType
VitalDelayArrayType01
VitalDelayArrayType01Z
VitalDelayArrayType01ZX

12.1.2 VITAL memory arc type

The VITAL Level 1 Memory timing specification defines a VitalMemoryArcType which represents the different timing arc relationships between two vector objects.

type VitalMemoryArcType **is** (ParallelArc , CrossArc, SubwordArc);

The enumerated values which describe these relationships are defined as follows and illustrated in Figure 12-1:

— ParallelArc represents a one-to-one timing relationship between two vector objects of the same size, wherein each sub-element of the first object is related only to the corresponding sub-element of the second object.
— CrossArc represents a timing relationship between two vector objects, wherein each sub-element of the first object is related to each sub-element of the second object.
— SubwordArc represents a timing relationship between two vector objects wherein each sub-element of the first vector object is related to a set of sub-elements of the second vector object as defined by a memory subword relationship (see [C11.1.1]).

 Copyright © 2001 IEEE. All rights reserved.

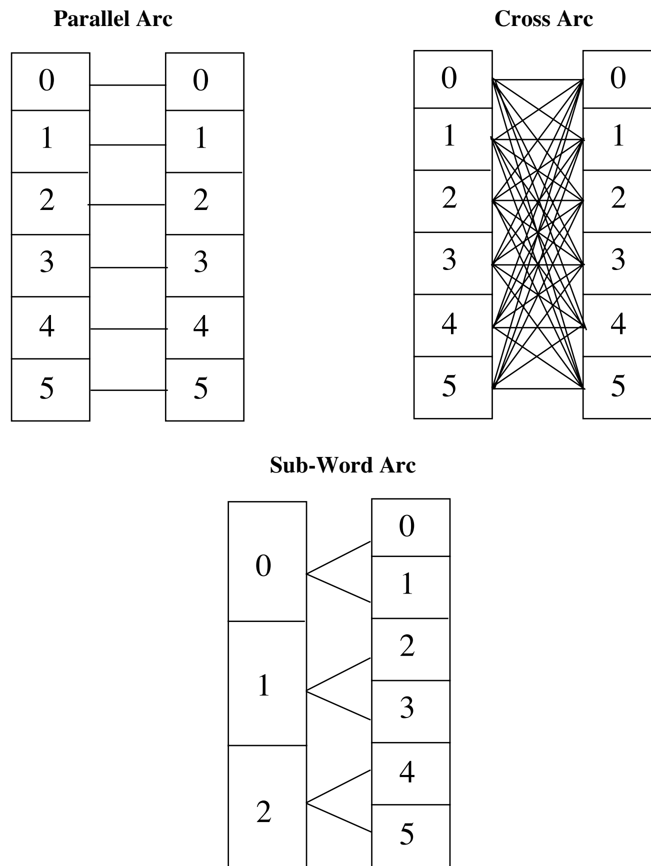


Figure 13—VITAL Memory Arc Types

NOTES:

1) A timing arc can represent a delay arc or a timing constraint arc.

2) The legal sizes of the timing generic between two vector signals of unequal sizes requires that the size of the timing generic be the product of the length of the first signal and the length of the second signal (see 4.3.2.1.2). As a result, the timing generic associated with a SubwordArc contains redundant sub-elements. The values corresponding to these sub-elements are not considered during path delay selection or timing constraint checks.

12.2 Memory Output Retain timing behavior

The VITAL Level 1 Memory specification supports modeling of output retain timing behavior observed in typical ASIC memories. For all timing arcs, any input change which causes a transition on the output, the Level 1 memory path delay procedures schedule the new value on the output signal after the specified propagation delay. If an output retain time is also specified for a timing arc, the following behavior is exhibited:

— The output signal retains its value for a time period equal to output retain time.
— The new value is scheduled on the output after the propagation delay.
— For the time period between the output retain time and the propagation delay, the output is set to ‘X’ value.

Copyright © 2001 IEEE. All rights reserved.

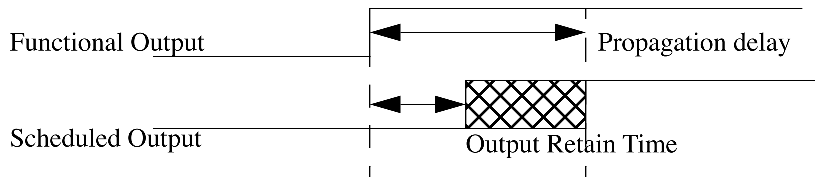


Figure 14—Output retain behavior

12.3 VITAL Memory output retain timing specification

The output retain time is introduced in a VITAL compliant memory model using the transition dependent delay types VitalDelayType01Z, VitalDelayType01ZX and their vector forms(see 5.x.x). These delay types hold the output retain timing values based on a mapping to a specific transition. Output retain timing values along with propagation delay values are used for appropriate delay selection by the VITAL Memory path delay procedures. The mapping of the transition dependent delay locations in the delay types VitalDelayType01Z and VitalDelayType01ZX to the output retain timing values is defined in Table 24.

Table 24—Delay locations used to propagation and output retain delay values

Delay type	tr01	tr10	tr0Z	trZ1	tr1Z	trZ0	tr0X	trX1	tr1X	trX0	trXZ	trZX
Vital Delay Type 01Z	prop. delay	prop. delay	output retain time	not used	output retain time	not used						
Vital Delay Type 01ZX	prop. delay	prop. delay	prop. delay	prop. delay	prop. delay	prop. delay	output retain time	not used	output retain time	not used	not used	output retain time

Similar is applicable to vector forms VitalDelayType01Z and VitalDelayType01ZX. A delay path can be specified to have output retain behavior by setting OutputRetainFlag to True in the VitalMemoryAddPathDelay procedure. The default value of OutputRetainFlag is False.

12.4 Transition dependent delay selection

Delay selection for a particular signal may be based upon the new and previous values of the signal. This selection mechanism is called *transition dependent delay selection*. Transitions between the previous and new values are described by enumeration values of the predefined type VitalTransitionType. The delay selection for paths without output retain timing behavior is explained in Table 7. The delay selection for paths with output retain delay behavior is shown in Table 25.

 Copyright © 2001 IEEE. All rights reserved.

Table 25—Transition dependent delay selection for paths with output retain behavior

Previous value	New value	Delay selected for VitalDelayType01Z	Delay selected for VitalDelayType01ZX
‘0’	‘1’	Delay(tr01)	Delay(tr01)
0	Z	D elay(tr01)	Delay(tr0Z)
0	X	D elay(tr01)	Min(Delay(tr01), Delay(tr0Z))
1	0	D elay(tr10)	Delay(tr10)
1	Z	D elay(tr10)	Delay(tr1Z)
1	X	D elay(tr10)	Min(Delay(tr10), Delay(tr1Z))
Z	0	D elay(tr10)	Delay(trZ0)
Z	1	D elay(tr01)	Delay(trZ1)
Z	X	Min(D elay(tr10), Delay(tr01))	Min(Delay(trZ0), Delay(trZ1))
X	0	D elay(tr10)	Max(Delay(tr10), Delay(trZ0))
X	1	D elay(tr01)	Max(Delay(tr01), Delay(trZ1))
X	Z	Max(D elay(tr01), Delay(tr10))	Max(Delay(tr0Z), Delay(tr1Z))

NOTES:

1) Transition dependent delay selection for paths with output retain behavior for delay type VitalDelayType01Z is same as VitalDelayType01 used in VitalPathDelay01 procedure.

2) Transition dependent delay selection for paths with output retain behavior for delay type VitalDelayType01ZX is same as VitalDelayType01Z used in VitalPathDelay01Z procedure.

12.5 VITAL memory path delay procedures

The VITAL Level 1 Memory timing specification provides three procedures to perform path delay selection and scheduling:

— VitalMemoryInitPathDelay
— VitalMemoryAddPathDelay
— VitalMemorySchedulePathDelay

These procedures support the following capabilities:

— Transition dependent path delay selection and scheduling of the functional output value on the specified signal.
— Condition dependent delay path selection. Both scalar and vector form of path conditions are supported.
— Support for the all the delay types as specified in 12.1.1.
— Support for output retain timing behavior.
— Support for ParallelArc, CrossArc and SubwordArc timing relationships, between input and output signals.
— Support for strength modification of the output value.
— Support for scalar and vector forms of output/internal signals and input variables.

Copyright © 2001 IEEE. All rights reserved.

For every output port or internal signal, the output delay selection and scheduling is performed using the VITAL memory path delay procedures as follows:

— The delay scheduling information is initialized through a call to VitalMemoryInitPathDelay.
— Each candidate path to that output is specified through a call to VitalMemoryAddPathDelay.
— The delay selected based on each candidate path is scheduled on the output through a call to VitalMemorySchedulePathDelay.

See [C10.4.1.2.3] for restrictions on the usage of these VITAL memory path delay procedures.

For every output change, selection of the appropriate path delay begins with the selection of candidate paths. The candidate paths are selected by identifying all the calls to VitalMemoryAddPathDelay for which the PathCondition is True. If there is a single candidate path, its delay is the one selected. If there is more than one candidate path, the shortest delay (accounting for the InputChangeTime) is selected using transition dependent delay selection. If there are no candidate paths, the output change is not scheduled and output remains unchanged. VHDL transport delay mode is used for delay scheduling and no glitch handling is perfomed in these procedures.

12.5.1 VitalMemoryInitPathDelay

The VitalMemoryInitPathDelay procedure is used to initialize the output path delay schedule data structure denoted by the ScheduleData formal parameter. The delay and schedule data structure consists of information related to propagation delays, current and previously scheduled times, current and previously scheduled values. VitalMemoryInitPathDelay procedure shall only be specified one per output port and supports overloaded scalar and vector forms. The Table 26 explains the formal arguments of the VitalMemoryInitPathDelay procedure:

Table 26—Formal arguments for VitalMemoryInitPathDelay

Argument name	Argument type	Argument description
ScheduleData	VitalMemoryScheduleDataType	Data structure used by VITAL memory path delay procedures to store persistent information for the path delay scheduling.
ScheduleDataArray	VitalMemoryScheduleDataVector-Type	Data structure used by VITAL memory path delay procedures to store persistent information for the path delay scheduling for vector outputs.
NumBitsPerSubWord	POSITIVE	Number of bits in one memory subword
OutputData	STD_ULOGIC	Scalar form of the functional output value to be scheduled.
OutputDataArray	STD_LOGIC_VECTOR	Vector form of the functional output value to be scheduled.

12.5.2 VitalMemoryAddPathDelay

The VitalMemoryAddPathDelay procedure is used to specify a delay path from an input to the output port or an internal signal. There shall be exactly one call to this procedure for each delay path from an input to an output or internal signal. During memory path delay selection process, the VitalMemoryAddPathDelay procedures are used for selecting candidate paths based on PathCondition or PathConditionArray being True. The ScheduleData or ScheduleDataArray structure is updated accordingly. The Table 27 explains the formal arguments of the VitalMemoryAddPathDelay procedure:

 Copyright © 2001 IEEE. All rights reserved.

Table 27—Formal arguments for VitalMemoryAddPathDelay

Argument name	Argument type	Argument description
ScheduleDatan	VitalMemoryScheduleDataType	Data structure used by VITAL memory path delay procedures to store persistent information for the path delay scheduling.
ScheduleDataArray	VitalMemoryScheduleDataVectorType	Data structure used by VITAL memory path delay procedures to store persistent information for the path delay scheduling for vector outputs.
InputSignal	STD_LOGIC_VECTOR	Array used to store input change values
InputChangeTime	Time	Array used to store the time since the last input change occured.
PathDelayArray	VitalDelayArrayType01Z	Array of path delay values used to propagate the output values.
PathCondition, PathConditionArray	BOOLEAN VitalBoolArrayT	Condition under which the delay path is considered to be one of the candidate paths for propagation delay selection.
OutputRetainFlag	BOOLEAN	If set to TRUE, the memory output retain behavior is enabled. Set to FALSE by default.
OutputRetainBehavior	OutputRetainBehaviorType	A value BitCorrupt/WordCorrupt for this flag indicates the output bits to be set to intermediate “X” value on a bit-by-bit/word basis respectively corresponding to each input bit change during retain propagation.
ArcType	VitalMemoryArcType	Delay arc type between input and output.

The following additional requirements are placed on the usage of PathConditionArray parameter:

— The mapping of various bits in the PathConditionArray parameter to OutputSignal is based on a SubWord relationship.
— The number of bits in OutputSignal controlled by each corresponding bit in the PathConditionArray parameter is specified by the NumBitsPerSubword parameter in the associated call to VitalMemoryInitPathDelay.

The VitalMemoryAddPathDelay procedure is overloaded to support the combinations of VITAL delay types, path delay parameters, input signals, output signals and path condition parameters in Table 28:

Copyright © 2001 IEEE. All rights reserved.

Table 28—Overloaded versions of VitalMemoryAddPathDelay

VITAL delay type	Input signal	Output signal	Path delay	Path condition
VitalDelayType	vector	vector	vector	scalar
VitalDelayType	scalar	vector	vector	scalar
VitalDelayType	vector	vector	vector	vector
VitalDelayType	scalar	vector	vector	vector
VitalDelayType	vector	scalar	vector	scalar
VitalDelayType	scalar	scalar	scalar	scalar
VitalDelayType01	vector	vector	vector	scalar
VitalDelayType01	scalar	vector	vector	scalar
VitalDelayType01	vector	vector	vector	vector
VitalDelayType01	scalar	vector	vector	vector
VitalDelayType01	vector	scalar	vector	scalar
VitalDelayType01	scalar	scalar	scalar	scalar
VitalDelayType01Z	vector	vector	vector	scalar
VitalDelayType01Z	scalar	vector	vector	scalar
VitalDelayType01Z	vector	vector	vector	vector
VitalDelayType01Z	scalar	vector	vector	vector
VitalDelayType01Z	vector	scalar	vector	scalar
VitalDelayType01Z	scalar	scalar	scalar	scalar
VitalDelayType01ZX	vector	vector	vector	scalar
VitalDelayType01ZX	scalar	vector	vector	scalar
VitalDelayType01ZX	vector	vector	vector	vector
VitalDelayType01ZX	scalar	vector	vector	vector
VitalDelayType01ZX	vector	scalar	vector	scalar
VitalDelayType01ZX	scalar	scalar	scalar	scalar

The overloaded versions of VitalMemoryAddPathDelay for which the delay type is VitalDelayType01Z or VitalDelayType01ZX and their vector forms support modeling of the memory output retain behavior.

12.5.3 VitalMemorySchedulePathDelay

The VitalMemorySchedulePathDelay procedure is used to schedule the functional output value on the output port or internal signal with the selected propagation delay. The VitalMemorySchedulePathDelay procedure is overloaded for scalar and vector outputs. The VitalMemorySchedulePathDelay procedure can be used to perform result mapping of the output value using the OutputMap parameter

 Copyright © 2001 IEEE. All rights reserved.

Table 29 explains the formal arguments of the VitalMemorySchedulePathDelay procedure:

Table 29—Formal arguments for VitalMemorySchedulePathDelay

Argument name	Argument type	Argument description
ScheduleData	VitalMemoryScheduleDataType	Data structure used by VITAL memory path delay procedures to store persistent information for the path delay scheduling.
ScheduleDataArray	VitalMemoryScheduleDataVectorType	Data structure used by VITAL memory path delay procedures to store persistent information for the path delay scheduling for vector outputs.
OutputSignal	STD_ULOGIC	Scalar form of output or internal signal being driven
OutputSignal	STD_LOGIC_VECTOR	Vector form of output or internal signal being driven
OutputSignalName	STRING	Name of output port or internal signal
OutputMap	VitalOutputMapType	Strength mapping of output values

Example:

This example illustrates the use of all three VITAL memory path delay procedures.

```
--- InitPathDelay call for the output bus DOA. Precedes all AddPathDelay calls for the output
DOA.
    VitalMemoryInitPathDelay (
        ScheduleDataArray      => DOA_ScheduleDataArray,
        OutputDataArray        => DOA_zd
        );
--- AddPathDelay call for the ADDR to DOA path. All AddPathDelay calls for a particular output
--- follow the InitPathDelay call for that particular output.
  VitalMemoryAddPathDelay (
        ScheduleDataArray      => DOA_ScheduleDataArray,
        InputSignal            => AADR_i,
        OutputSignalName       => "DOA",
        InputChangeTimeArray   => AADR_InputChangeArray,
        PathCondition          => TRUE,
        PathDelayArray         => AADR_DOA_Delay,
        ArcType                => CrossArc              --Timing path is a cross arc
        );

-- AddPathDelay call for the DataIn (DIC) to DataOut  (DOA) path.
  VitalMemoryAddPathDelay (
        ScheduleDataArray      => DOA_ScheduleDataArray,
        InputSignal            => DIC_dly,
        OutputSignalName       => "DOA",
        InputChangeTimeArray   => DIC_InputChangeArray,
        PathDelayArray         => DIC_DOA_Delay,
        PathCondition          => TRUE,
        ArcType                =>  ParallelArc          --Timing path is a parallel arc
        );
```

Copyright © 2001 IEEE. All rights reserved.

```
-- SchedulePathDelay call follows after the last AddPathDelay call for a particular output
   VitalMemorySchedulePathDelay (
          OutSignal              => DOA_i,
          OutputSignalName       => "DOA",
          OutputMap              => VitalDefaultOutputMap,
          ScheduleDataArray      => DOA_ScheduleDataArray
     );
```

12.6 VITAL memory timing check procedures

The Level 1 Memory timing specification defines two timing check procedures: VitalMemorySetupHoldCheck and VitalMemoryPeriodPulseCheck. A Level 1 VITAL memory model can use the timing check procedures declared in the VITAL_Timing package. These timing check procedures perform the following functions:

— Detect a timing constraint violation if the timing check is enabled.
— Report a timing constraint violation using a VHDL assertion statement. The report message and severity level of the assertion are controlled by the model.
— Sets the value of a corresponding violation flag(s) to 'X' upon detection of a timing violation is, otherwise it is set to '0'. 'X' generation for this flag(s) can be controlled by the model.

The same timing check procedures are used for both positive and negative timing constraint values. Two delay parameters —TestDelay and RefDelay — are defined for modeling the delays associated with the test or reference signals when negative setup or hold constraints are in effect. The delay parameters shall have the value of zero when negative constraints do not apply.

12.6.1 VitalMemorySetupHoldCheck

The procedure VitalMemorySetupHoldCheck detects the presence of a setup or hold violation on the input test signal with respect to the corresponding input reference signal. The input test and reference signals can be a scalar or vector signal. The timing constraints are specified through parameters representing the high and low values for the setup and hold times. This procedure assumes non-negative values for setup/hold timing constraints.

Setup/hold constraint checks are performed by this procedure only if the CheckEnabled condition evaluates to True; however, event times required for constraint checking are always updated, regardless of the value of CheckEnabled. In addition, four parameters are provided for finer control over condition checking. A setup check will be performed only if EnableSetupOnTest is True at the test edge and EnableSetupOnRef is True at the reference edge. Similarly, a hold check will be performed only if EnableHoldOnTest is True at the test edge and EnableHoldOnRef is True at the reference edge. Setup constraints are checked in the simulation cycle in which the reference edge occurs. A setup violation is detected if the time since the last TestSignal change is less than the expected setup constraint time. Hold constraints are checked in the simulation cycle in which an event on TestSignal occurs. A hold violation is detected if the time since the last reference edge is less than the expected hold constraint time.

VitalMemorySetupHoldCheck provides the following capabilities:

— Support for ParallelArc, CrossArc and SubwordArc timing relationships between the input test signal and the input reference signal.
— Support for vector violation flags in Violation parameter.
— Support for scalar and vector forms of condition specification in timing checks using CheckEnabled.

 Copyright © 2001 IEEE. All rights reserved.

— Supports MsgFormat parameter which can be used to control the format of the test/reference signals in the message. The test/reference signals can be reported as either vectors or as scalars.

Table 30 describes some of the formal arguments for the VitalMemorySetupHoldCheck procedure.

Table 30—Formal arguments for VitalMemorySetupHoldCheck

Argument name	Argument type	Argument description
Violation	X01ArrayT	Vector only form of violation flags.
TimingData	VitalMemoryTiming-DataType	Data structure used by VITAL memory setup/hold check procedure to store persistent information for checking timing constraint violations.
TestSignal	STD_LOGIC_VECTOR, STD_ULOGIC	Scalar and vector forms of test signal
TestDelay	VitalDelayType, VitalDelayArrayType	Internal delay for the test port (scalar and vector forms)
RefSignal	STD_LOGIC_VECTOR,STD_ULOGIC	Reference Signal
RefDelay	VitalDelayType, VitalDelayArrayType	Internal delay for the reference port (scalar and vector forms)
SetupHigh, SetupLow, HoldHigh, HoldLow	VitalDelayType, VitalDelayArrayType	Setup and Hold check values. No default values are provided for these parameters unlike in the case of VitalSetupHoldCheck.
CheckEnabled	BOOLEAN, VitalBoolArrayT	Condition under which the timing check should be performed
RefTransition	VitalEdgeSymbolType	Transition of the reference signal w.r.t which the timing check is done.
ArcType	VitalMemoryArcType	Timing Check Arc type
NumBitsPerSubWord	INTEGER	Number of bits per sub-word
MsgFormat	VitalMemoryMsgFormatType	To control the format of messages - Scalar, vector or vector enum

VitalMemorySetupHoldCheck procedure is overloaded for the cases shown in Table 31:

Table 31—Overload versions of VitalMemorySetupHoldCheck

Input Test signal	Input Reference signal	CheckEnabled
Scalar	Scalar	Vector
Vector	Scalar	Scalar
Vector	Scalar	Vector
Vector	Vector	Scalar
Vector	Vector	Vector

Copyright © 2001 IEEE. All rights reserved.

Table 32 captures the legal sizes of Violation for all overloaded forms of VitalMemorySetupHoldCheck:

Table 32—Legal sizes of violation flags

TestSignal	RefSignal	CheckEnabled	ArcType	Violation Size
Scalar	Scalar	Vector	N/A	Size of CheckEnabled
Vector	Scalar	Scalar	N/A	Size of TestSignal
Vector	Scalar	Vector	CrossArc relation between TestSignal and CheckEnabled	Size of CheckEnabled
Vector	Scalar	Vector	SubwordArc or ParallelArc relation between TestSignal and CheckEnabled	Size of CheckEnabled Or Size of TestSignal
Vector Vector	Vector Vector	Scalar Vector	CrossArc relation between TestSignal and RefSignal	Size of RefSignal
Vector Vector	Vector Vector	Scalar Vector	SubwordArc or ParallelArc relation between TestSignal and RefSignal	Size of RefSignal Or Size of TestSignal

NOTES:

1) In the overloaded version of VitalMemorySetupHoldCheck, where TestSignal is a vector, RefSignal is a scalar and CheckEnabled is a vector, the value of ArcType and NumBitsPerSubWord is used to determine the relationship between the sub-elements of TestSignal and CheckEnabled parameters.

2) In the overloaded version of VitalMemorySetupHoldCheck, where TestSignal, RefSignal and CheckEnabled all assume vector forms, the value of NumBitsPerSubWord is used to determine the relationship between the sub-elements of TestSignal and CheckEnabled parameters.

12.6.2 VitalMemoryPeriodPulseCheck

The procedure VitalMemoryPeriodPulseCheck checks for minimum and maximum periodicity and pulse width for '1' and '0' values of the input test vector signal. The timing constraint is specified through parameters representing the minimal period between successive rising or falling edges of the input test signal, and the minimum pulse widths associated with high and low values. The format of the message given in case of a violation can be controlled using the MsgFormat parameter. The VitalMemoryPeriodPulseCheck procedure is overloaded for scalar and vector versions of Violation flags.

Table 33 describes some of the formal arguments for the VitalMemorySetupHoldCheck procedure.

Table 33—Formal arguments for VitalMemorySetupHoldCheck

Argument name	Argument type	Argument description
Violation	X01ArrayT, X01	Vector and scalar forms of violation flags.
PeriodData	VitalPeriodDataArrayType	Data structure used by VITAL memory period/pulse check procedure to store persistent information for checking timing constraint violations.
TestSignal	STD_LOGIC_VECTOR	Vector form of test signal

 Copyright © 2001 IEEE. All rights reserved.

Table 33—Formal arguments for VitalMemorySetupHoldCheck (continued)

Argument name	Argument type	Argument description
TestDelay	VitalDelayArrayType	Internal delay for the test port
PulseWidthHigh PulseWidthLow	VitalDelayArrayType	Pulse width values. No default values are provided for these parameters unlike in the case of VitalPeriodPulseCheck.
Period	VitalDelayArrayType	Period values. No default values are provided for these parameters unlike in the case of VitalPeriodPulseCheck.
CheckEnabled	BOOLEAN	Condition under which the timing check should be performed.
MsgFormat	VitalMemoryMsgFormatType	To control the format of messages - Scalar, vector or vector enum

Example:

```
    MemBehavior : PROCESS ( WEC_i, OE_i, DIC_i,  CADR_i )
      VARIABLE TimingDataInfo_DIC_WEC :
            VitalMemoryTimingDataType :=     VitalMemoryTimingDataInit;       --  Restricted
variable

      VARIABLE TimingDataInfo_CADR_WECR :
            VitalMemoryTimingDataType :=VitalMemoryTimingDataInit;  -- Restricted variable

      VARIABLE TimingDataInfo_CADR_WECF :
              VitalMemoryTimingDataType  :=  VitalMemoryTimingDataInit;      --  Restricted
variable

      VARIABLE PeriodDataInfo_WEC :
            VitalMemoryPeriodDataType  :=    VitalMemoryPeriodDataInit  ;       --  Restricted
variable

      VARIABLE Viol_DIC_WEC : X01ArrayT(3 DOWNTO 0) := (others => '0');
      VARIABLE Viol_CADR_WECF ,Viol_CADR_WECR,Viol_WEC      : X01 := '0';
      ....
      BEGIN
      IF ( TimingChecksOn) then

            --- Setup/Hold check between D and falling WEC. SubwordArc timing check arc
             VitalMemorySetupHoldCheck (
                    Violation                 => Viol_DIC_WEC,
                    TimingData                => TimingDataInfo_DIC_WEC,
                    TestSignal                => DIC_dly,
                    TestSignalName            => "DIC",
                    TestDelay                 => tisd_DIC_WEC,
                    RefSignal                 => WEC_dly,
                    RefSignalName             => "WEC",
                    RefDelay                  => ticd_WEC,
                    SetupHigh                 => tsetup_DIC_WEC_posedge_negedge,
                    Setuplow                  => tsetup_DIC_WEC_negedge_negedge,
                    HoldHigh                  => thold_DIC_WEC_negedge_negedge,
```

Copyright © 2001 IEEE. All rights reserved.

```
                HoldLow              => thold_DIC_WEC_posedge_negedge,
                ArcType              =>  SubwordArc, --Timing arc is of subword type
                NumBitsPerSubword    => 2,            -- Number of bits per subword.
                CheckEnabled         => TRUE,
                RefTransition        => '\',
                HeaderMsg            => "*",
                MsgOn                => MsgOn,
                XOn                  => XOn,
                MsgSeverity          => MsgSeverity,
                MsgFormat            => Vector
            );
        --- Setup/Hold check between CADR and falling WEC. CrossArc timing check arc
          VitalMemorySetupHoldCheck (
                Violation            => Viol_CADR_WECF,
                TimingData           => TimingDataInfo_CADR_WECF,
                TestSignal           => CADR_dly,
                TestSignalName       => "CADR",
                TestDelay            => tisd_CADR_WEC,
                RefSignal            => WEC_dly,
                RefSignalName        => "WEC",
                RefDelay             => ticd_WEC,
                SetupHigh            => tsetup_CADR_WEC_posedge_negedge,
                Setuplow             => tsetup_CADR_WEC_negedge_negedge,
                HoldHigh             => thold_CADR_WEC_posedge_negedge,
                HoldLow              => thold_CADR_WEC_negedge_negedge,
                CheckEnabled         => TRUE,
                RefTransition        => '\',
                HeaderMsg            => "*",
                MsgOn                => MsgOn,
                XOn                  => XOn,
                MsgSeverity          => MsgSeverity,
                MsgFormat            => Vector
            );
        --- PulseWidth Low Check for WEC. Period and Pulse Width High values need to
be
        --- set to 0 ns since no default values are set by the procedure

        VitalMemoryPeriodPulseCheck (
               Violation             =>  Viol_WEC,
               PeriodData            =>  PeriodDataInfo_WEC,
               TestSignal            =>   WEC_i,
               TestSignalName        =>   “WEC”,
               TestDelay             =>   ticd_WEC,
               Period                =>    ( 0 ns , 0 ns ),
               PulseWidthHigh        =>    ( 0 ns , 0 ns ),
               PulseWidthLow         =>     tpw_WEC_negedge,
               CheckEnabled          => TRUE,
               HeaderMsg             => "*",
               MsgOn                 => MsgOn,
               XOn                   => XOn,
               MsgSeverity           => MsgSeverity,
               MsgFormat             => Vector
        );
    END IF;
```

 Copyright © 2001 IEEE. All rights reserved.

13. The VITAL standard packages

The VITAL ASIC modeling specification defines two standard packages VITAL_Timing and VITAL_Primitives which predefine a number of items that are useful or required for designing VITAL compliant models. These packages reside in the VHDL library IEEE.

The semantics of the VITAL standard packages are defined by their VHDL description according to the IEEE Std 1076-1987 and the IEEE Std 1164. Their interfaces are defined by their package declarations and their behavior is defined by the corresponding package bodies. An implementation may not add items, delete items, or otherwise alter the contents of the VITAL standard packages. An implementation may choose to implement the package bodies in a more efficient form; however, the resulting semantic shall not differ from the formal semantic provided herein.

13.1 VITAL_Timing package declaration

```
-------------------------------------------------------------------------------
-- Title        : Standard VITAL TIMING Package
--              : $Revision: 1.3 $
--              :
-- Library      : This package shall be compiled into a library
--              : symbolically named IEEE.
--              :
-- Developers   : IEEE DASC Timing Working Group (TWG), PAR 1076.4
--              :
-- Purpose      : This packages defines standard types, attributes, constants,
--              : functions and procedures for use in developing ASIC models.
--              :
-- Known Errors :
--              :
-- Note         : No declarations or definitions shall be included in,
--              : or excluded from this package. The "package declaration"
--              : defines the objects (types, subtypes, constants, functions,
--              : procedures ... etc.) that can be used by a user.  The package
--              : body shall be considered the formal definition of the
--              : semantics of this package.  Tool developers may choose to
--              : implement the package body in the most efficient manner
--              : available to them.
-- ----------------------------------------------------------------------------
--
-- ----------------------------------------------------------------------------
-- Acknowledgments:
--   This code was originally developed under the "VHDL Initiative Toward ASIC
--   Libraries" (VITAL), an industry sponsored initiative.  Technical
--   Director: William Billowitch, VHDL Technology Group; U.S. Coordinator:
--   Steve Schultz;  Steering Committee Members: Victor Berman, Cadence Design
--   Systems; Oz Levia, Synopsys Inc.; Ray Ryan, Ryan & Ryan; Herman van Beek,
--   Texas Instruments; Victor Martin, Hewlett-Packard Company.
-- ----------------------------------------------------------------------------
--
-- ----------------------------------------------------------------------------
-- Modification History :
-- ----------------------------------------------------------------------------
-- Version No:|Auth:| Mod.Date:| Changes Made:
--   v95.0 A  |     | 06/02/95 | Initial ballot draft 1995
--   v95.1    |     | 08/31/95 | #203 - Timing violations at time 0
--                               #204 - Output mapping prior to glitch detection
--   v98.0    |TAG  | 03/27/98 | Initial ballot draft 1998
--                             | #IR225 - Negative Premptive Glitch
--                                      **Pkg_effected=VitalPathDelay,
--                                      VitalPathDelay01,VitalPathDelay01z.
--                               #IR105 - Skew timing check needed
--                                      **Pkg_effected=NONE, New code added!!
--                               #IR248 - Allows VPD to use a default timing
--                                        delay
--                                      **Pkg_effected=VitalPathDelay,
--                                        VitalPathDelay01,VitalPathDelay01z,
--                          #IR250 - Corrects fastpath condition in VPD
--                                      **Pkg_effected=VitalPathDelay01,
--                                        VitalPathDelay01z,
--                               #IR252 - Corrects cancelled timing check call if
--                                        condition expires.
--                                        **Pkg_effected=VitalSetupHoldCheck,
```

Copyright © 2001 IEEE. All rights reserved.

```
--                                      VitalRecoveryRemovalCheck.
--                             #IR105 - Skew timing check
--                                      **Pkg_effected=NONE, New code added
--   v98.1    | jdc | 03/25/99 | Changed UseDefaultDelay to IgnoreDefaultDelay
--                             and set default to FALSE in VitalPathDelay()
--   v00.7    | dbb | 07/18/00 | Removed "maximum" from VitalPeriodPulse()
--                             comments


LIBRARY IEEE;
USE     IEEE.Std_Logic_1164.ALL;

PACKAGE VITAL_Timing IS
    TYPE VitalTransitionType IS ( tr01, tr10, tr0z, trz1, tr1z, trz0,
                                  tr0X, trx1, tr1x, trx0, trxz, trzx);

    SUBTYPE VitalDelayType     IS TIME;
    TYPE VitalDelayType01   IS ARRAY (VitalTransitionType   RANGE tr01 to tr10)
         OF TIME;
    TYPE VitalDelayType01Z  IS ARRAY (VitalTransitionType   RANGE tr01 to trz0)
         OF TIME;
    TYPE VitalDelayType01ZX IS ARRAY (VitalTransitionType   RANGE tr01 to trzx)
         OF TIME;

    TYPE VitalDelayArrayType     IS ARRAY (NATURAL RANGE <>) OF VitalDelayType;
    TYPE VitalDelayArrayType01   IS ARRAY (NATURAL RANGE <>) OF VitalDelayType01;
    TYPE VitalDelayArrayType01Z  IS ARRAY (NATURAL RANGE <>) OF VitalDelayType01Z;
    TYPE VitalDelayArrayType01ZX IS ARRAY (NATURAL RANGE <>) OF VitalDelayType01ZX;
    -- ----------------------------------------------------------------------
    -- **********************************************************************
    -- ----------------------------------------------------------------------

    CONSTANT VitalZeroDelay     : VitalDelayType     :=   0 ns;
    CONSTANT VitalZeroDelay01   : VitalDelayType01   := ( 0 ns, 0 ns );
    CONSTANT VitalZeroDelay01Z  : VitalDelayType01Z  := ( OTHERS => 0 ns );
    CONSTANT VitalZeroDelay01ZX : VitalDelayType01ZX := ( OTHERS => 0 ns );

    ---------------------------------------------------------------------------
    -- examples of usage:
    ---------------------------------------------------------------------------
    -- tpd_CLK_Q : VitalDelayType  := 5 ns;
    -- tpd_CLK_Q : VitalDelayType01  := (tr01 => 2 ns, tr10 => 3 ns);
    -- tpd_CLK_Q : VitalDelayType01Z := ( 1 ns, 2 ns, 3 ns, 4 ns, 5 ns, 6 ns );
    -- tpd_CLK_Q : VitalDelayArrayType(0 to 1)
    --                    := (0 => 5 ns, 1 => 6 ns);
    -- tpd_CLK_Q : VitalDelayArrayType01(0 to 1)
    --                    := (0 => (tr01 => 2 ns, tr10 => 3 ns),
    --                        1 => (tr01 => 2 ns, tr10 => 3 ns));
    -- tpd_CLK_Q : VitalDelayArrayType01Z(0 to 1)
    --                  := (0 => ( 1 ns, 2 ns, 3 ns, 4 ns, 5 ns, 6 ns ),
    --                      1 => ( 1 ns, 2 ns, 3 ns, 4 ns, 5 ns, 6 ns ));
    ---------------------------------------------------------------------------

    -- TRUE if the model is LEVEL0 | LEVEL1 compliant
    ATTRIBUTE VITAL_Level0 : BOOLEAN;
    ATTRIBUTE VITAL_Level1 : BOOLEAN;

    SUBTYPE std_logic_vector2 IS std_logic_vector(1 DOWNTO 0);
    SUBTYPE std_logic_vector3 IS std_logic_vector(2 DOWNTO 0);
    SUBTYPE std_logic_vector4 IS std_logic_vector(3 DOWNTO 0);
    SUBTYPE std_logic_vector8 IS std_logic_vector(7 DOWNTO 0);

    -- Types for strength mapping of outputs
    TYPE VitalOutputMapType  IS ARRAY ( std_ulogic ) OF std_ulogic;
    TYPE VitalResultMapType  IS ARRAY ( UX01       ) OF std_ulogic;
    TYPE VitalResultZMapType IS ARRAY ( UX01Z      ) OF std_ulogic;
    CONSTANT VitalDefaultOutputMap  : VitalOutputMapType
                                    := "UX01ZWLH-";
    CONSTANT VitalDefaultResultMap  : VitalResultMapType
                                    := ( 'U', 'X', '0', '1' );
    CONSTANT VitalDefaultResultZMap : VitalResultZMapType
                                    := ( 'U', 'X', '0', '1', 'Z' );

    -- Types for fields of VitalTimingDataType
    TYPE VitalTimeArrayT  IS ARRAY (INTEGER RANGE <>) OF TIME;
    TYPE VitalTimeArrayPT IS ACCESS VitalTimeArrayT;
    TYPE VitalBoolArrayT  IS ARRAY (INTEGER RANGE <>) OF BOOLEAN;
    TYPE VitalBoolArrayPT IS ACCESS VitalBoolArrayT;
    TYPE VitalLogicArrayPT IS ACCESS std_logic_vector;

    TYPE VitalTimingDataType IS RECORD
```

 Copyright © 2001 IEEE. All rights reserved.

```
    NotFirstFlag : BOOLEAN;
    RefLast   : X01;
    RefTime   : TIME;
    HoldEn    : BOOLEAN;
    TestLast  : std_ulogic;
    TestTime  : TIME;
    SetupEn   : BOOLEAN;
    TestLastA : VitalLogicArrayPT;
    TestTimeA : VitalTimeArrayPT;
    HoldEnA   : VitalBoolArrayPT;
    SetupEnA  : VitalBoolArrayPT;
END RECORD;

FUNCTION VitalTimingDataInit RETURN VitalTimingDataType;

-- type for internal data of VitalPeriodPulseCheck
TYPE VitalPeriodDataType IS RECORD
    Last : X01;
    Rise : TIME;
    Fall : TIME;
    NotFirstFlag : BOOLEAN;
END RECORD;
CONSTANT VitalPeriodDataInit : VitalPeriodDataType
                             := ('X', 0 ns, 0 ns, FALSE );

-- Type for specifying the kind of Glitch handling to use
TYPE VitalGlitchKindType IS (OnEvent,
                             OnDetect,
                             VitalInertial,
                             VitalTransport);

TYPE VitalGlitchDataType IS
  RECORD
    SchedTime    : TIME;
    GlitchTime   : TIME;
    SchedValue   : std_ulogic;
    LastValue    : std_ulogic;
  END RECORD;
TYPE VitalGlitchDataArrayType IS ARRAY (NATURAL RANGE <>)
     OF VitalGlitchDataType;

-- PathTypes: for handling simple PathDelay info
TYPE VitalPathType IS RECORD
    InputChangeTime : TIME;            -- timestamp for path input signal
    PathDelay       : VitalDelayType;  -- delay for this path
    PathCondition   : BOOLEAN;         -- path sensitize condition
END RECORD;
TYPE VitalPath01Type IS RECORD
    InputChangeTime : TIME;            -- timestamp for path input signal
    PathDelay       : VitalDelayType01; -- delay for this path
    PathCondition   : BOOLEAN;         -- path sensitize condition
END RECORD;
TYPE VitalPath01ZType IS RECORD
    InputChangeTime : TIME;            -- timestamp for path input signal
    PathDelay       : VitalDelayType01Z;-- delay for this path
    PathCondition   : BOOLEAN;         -- path sensitize condition
END RECORD;

-- For representing multiple paths to an output
TYPE VitalPathArrayType    IS ARRAY (NATURAL RANGE <> ) OF VitalPathType;
TYPE VitalPathArray01Type  IS ARRAY (NATURAL RANGE <> ) OF VitalPath01Type;
TYPE VitalPathArray01ZType IS ARRAY (NATURAL RANGE <> ) OF VitalPath01ZType;

TYPE VitalTableSymbolType IS (
   '/',      -- 0 -> 1
   '\',      -- 1 -> 0
   'P',      -- Union of '/' and '^' (any edge to 1)
   'N',      -- Union of '\' and 'v' (any edge to 0)
   'r',      -- 0 -> X
   'f',      -- 1 -> X
   'p',      -- Union of '/' and 'r' (any edge from 0)
   'n',      -- Union of '\' and 'f' (any edge from 1)
   'R',      -- Union of '^' and 'p' (any possible rising edge)
   'F',      -- Union of 'v' and 'n' (any possible falling edge)
   '^',      -- X -> 1
   'v',      -- X -> 0
   'E',      -- Union of 'v' and '^' (any edge from X)
  'A',     -- Union of 'r' and '^' (rising edge to or from 'X')
  'D',     -- Union of 'f' and 'v' (falling edge to or from 'X')
   '*',      -- Union of 'R' and 'F' (any edge)
   'X',      -- Unknown level
```

```
        '0',      -- low level
        '1',      -- high level
        '-',      -- don't care
        'B',      -- 0 or 1
        'Z',      -- High Impedance
        'S'       -- steady value
    );

    SUBTYPE VitalEdgeSymbolType IS VitalTableSymbolType RANGE '/' TO '*';



-- Addition of Vital Skew Type Information
-- March 14, 1998

    ---------------------------------------------------------------------------
    -- Procedures and Type Definitions for Defining Skews
    ---------------------------------------------------------------------------

    TYPE VitalSkewExpectedType IS (none, s1r, s1f, s2r, s2f);

    TYPE VitalSkewDataType IS RECORD
        ExpectedType : VitalSkewExpectedType;
        Signal1Old1 : TIME;
        Signal2Old1 : TIME;
        Signal1Old2 : TIME;
        Signal2Old2 : TIME;
    END RECORD;

    CONSTANT VitalSkewDataInit : VitalSkewDataType := ( none, 0 ns, 0 ns, 0 ns, 0 ns );


    -- ------------------------------------------------------------------------
    --
    -- Function Name:   VitalExtendToFillDelay
    --
    -- Description:     A six element array of delay values of type
    --                  VitalDelayType01Z is returned when a 1, 2 or 6
    --                  element array is given.  This function will convert
    --                  VitalDelayType and VitalDelayType01 delay values into
    --                  a VitalDelayType01Z type following these rules:
    --
    --                  When a VitalDelayType is passed, all six transition
    --                  values are assigned the input value.  When a
    --                  VitalDelayType01 is passed, the 01 transitions are
    --                  assigned to the 01, 0Z and Z1 transitions and the 10
    --                  transitions are assigned to 10, 1Z and Z0 transition
    --                  values.  When a VitalDelayType01Z is passed, the values
    --                  are kept as is.
    --
    --                  The function is overloaded based on input type.
    --
    --                  There is no function to fill a 12 value delay
    --                  type.
    --
    -- Arguments:
    --
    --  IN                 Type           Description
    --                     Delay          A one, two or six delay value Vital-
    --                                    DelayType is passed and a six delay,
    --                                    VitalDelayType01Z, item is returned.
    --
    --  INOUT
    --   none
    --
    --  OUT
    --   none
    --
    --  Returns
    --   VitalDelayType01Z
    --
    -- ------------------------------------------------------------------------
    FUNCTION VitalExtendToFillDelay (
            CONSTANT Delay : IN VitalDelayType
          ) RETURN VitalDelayType01Z;
    FUNCTION VitalExtendToFillDelay (
            CONSTANT Delay : IN VitalDelayType01
          ) RETURN VitalDelayType01Z;
    FUNCTION VitalExtendToFillDelay (
            CONSTANT Delay : IN VitalDelayType01Z
```

 Copyright © 2001 IEEE. All rights reserved.

```
      ) RETURN VitalDelayType01Z;

-- ------------------------------------------------------------------------
--
-- Function Name:   VitalCalcDelay
--
-- Description:     This function accepts a 1, 2 or 6 value delay and
--                  chooses the correct delay time to delay the NewVal
--                  signal.  This function is overloaded based on the
--                  delay type passed. The function returns a single value
--                  of time.
--
--                  This function is provided for Level 0 models in order
--                  to calculate the delay which should be applied
--                  for the passed signal. The delay selection is performed
--                  using the OldVal and the NewVal to determine the
--                  transition to select.  The default value of OldVal is X.
--
--                  This function cannot be used in a Level 1 model since
--                  the VitalPathDelay routines perform the delay path
--                  selection and output driving function.
--
-- Arguments:
--
--  IN                 Type           Description
--                      NewVal         New value of the signal to be
--                                     assigned
--                      OldVal         Previous value of the signal.
--                                     Default value is 'X'
--                      Delay          The delay structure from which to
--                                     select the appropriate delay.  The
--                                     function overload is based on the
--                                     type of delay passed.  In the case of
--                                     the single delay, VitalDelayType, no
--                                     selection is performed, since there
--                                     is only one value to choose from.
--                                     For the other cases, the transition
--                                     from the old value to the new value
--                                     decide the value returned.
--
--  INOUT
--   none
--
--  OUT
--   none
--
--  Returns
--   Time                              The time value selected from the
--                                     Delay INPUT is returned.
--
-- -------------------------------------------------------------------------
FUNCTION VitalCalcDelay (
        CONSTANT NewVal : IN std_ulogic   := 'X';
        CONSTANT OldVal : IN std_ulogic   := 'X';
        CONSTANT Delay  : IN VitalDelayType
      ) RETURN TIME;
FUNCTION VitalCalcDelay (
        CONSTANT NewVal : IN std_ulogic   := 'X';
        CONSTANT OldVal : IN std_ulogic   := 'X';
        CONSTANT Delay  : IN VitalDelayType01
      ) RETURN TIME;
FUNCTION VitalCalcDelay (
        CONSTANT NewVal : IN std_ulogic   := 'X';
        CONSTANT OldVal : IN std_ulogic   := 'X';
        CONSTANT Delay  : IN VitalDelayType01Z
      ) RETURN TIME;

-- ------------------------------------------------------------------------
--
-- Function Name:   VitalPathDelay
--
-- Description:     VitalPathDelay is the Level 1 routine used to select
--                  the propagation delay path and schedule a new output
--                  value.
--
--                  For single and dual delay values, VitalDelayType and
--                  VitalDelayType01 are used.  The output value is
--                  scheduled with a calculated delay without strength
--                  modification.
--
--                  For the six delay value, VitalDelayType01Z, the output
```

```
--                  value is scheduled with a calculated delay.  The drive
--                  strength can be modified to handle weak signal strengths
--                  to model tri-state devices, pull-ups and pull-downs as
--                  an example.
--
--                  The correspondence between the delay type and the
--                  path delay function is as follows:
--
--                  Delay Type                Path Type
--
--                  VitalDelayType            VitalPathDelay
--                  VitalDelayType01          VitalPathDelay01
--                  VitalDelayType01Z         VitalPathDelay01Z
--
--                  For each of these routines, the following capabilities
--                  is provided:
--
--                  o Transition dependent path delay selection
--                  o User controlled glitch detection with the ability
--                    to generate "X" on output and report the violation
--                  o Control of the severity level for message generation
--                  o Scheduling of the computed values on the specified
--                    signal.
--
--                  Selection of the appropriate path delay begins with the
--                  candidate paths.  The candidate paths are selected by
--                  identifying the paths for which the PathCondition is
--                  true.  If there is a single candidate path, then that
--                  delay is selected.  If there is more than one candidate
--                  path, then the shortest delay is selected using
--                  transition dependent delay selection.  If there is no
--                  candidate paths, then the delay specified by the
--                  DefaultDelay parameter to the path delay is used.
--
--                  Once the delay is known, the output signal is then
--                  scheduled with that delay.  In the case of
--                  VitalPathDelay01Z, an additional result mapping of
--                  the output value is performed before scheduling.  The
--                  result mapping is performed after transition dependent
--                  delay selection but before scheduling the final output.
--
--                  In order to perform glitch detection, the user is
--                  obligated to provide a variable of VitalGlitchDataType
--                  for the propagation delay functions to use.  The user
--                  cannot modify or use this information.
--
-- Arguments:
--
--  IN              Type                    Description
--   OutSignalName  string                   The name of the output signal
--   OutTemp        std_logic                The new output value to be driven
--   Paths          VitalPathArrayType       A list of paths of VitalPathArray
--                  VitalPathArrayType01     type.  The VitalPathDelay routine
--                  VitalPathArrayType01Z    is overloaded based on the type
--                                           of constant passed in.  With
--                                           VitalPathArrayType01Z, the
--                                           resulting output strengths can be
--                                           mapped.
--   DefaultDelay   VitalDelayType           The default delay can be changed
--                  VitalDelayType01         from zero-delay to another set
--                  VitalDelayType01Z        of values.
--
--   IgnoreDefaultDelay BOOLEAN              If TRUE, the default delay will
--                                           be used when no paths are
--                                           selected.  If false, no event
--                                           will be scheduled if no paths are
--                                           selected.
--
--   Mode           VitalGlitchKindType      The value of this constant
--                                           selects the type of glitch
--                                           detection.
--                       OnEvent             Glitch on transition event
--                     | OnDetect            Glitch immediate on detection
--                     | VitalInertial       No glitch, use INERTIAL
--                                           assignment
--                     | VitalTransport      No glitch, use TRANSPORT
--                                           assignment
--   XOn            BOOLEAN                  Control for generation of 'X' on
--                                           glitch.  When TRUE, 'X's are
--                                           scheduled for glitches, otherwise
--                                           no are generated.
```

 Copyright © 2001 IEEE. All rights reserved.

```
--   MsgOn          BOOLEAN                Control for message generation on
--                                         glitch detect.  When TRUE,
--                                         glitches are reported, otherwise
--                                         they are not reported.
--   MsgSeverity    SEVERITY_LEVEL         The level at which the message,
--                                         or assertion, will be reported.
--   IgnoreDefaultDelay BOOLEAN            Tells the VPD whether to use the
--                                         default delay value in the absense
--                                 of a valid delay for input conditions 3/14/98 MG
--
--   OutputMap      VitalOutputMapType     For VitalPathDelay01Z, the output
--                                         can be mapped to alternate
--                                         strengths to model tri-state
--                                         devices, pull-ups and pull-downs.
--
--  INOUT
--   GlitchData     VitalGlitchDataType    The internal data storage
--                                         variable required to detect
--                                         glitches.
--
--  OUT
--   OutSignal      std_logic              The output signal to be driven
--
--  Returns
--   none
--
-- -------------------------------------------------------------------------
PROCEDURE VitalPathDelay (
    SIGNAL   OutSignal          : OUT   std_logic;
    VARIABLE GlitchData         : INOUT VitalGlitchDataType;
    CONSTANT OutSignalName      : IN    string;
    CONSTANT OutTemp            : IN    std_logic;
    CONSTANT Paths              : IN    VitalPathArrayType;
    CONSTANT DefaultDelay       : IN    VitalDelayType      := VitalZeroDelay;
    CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
    CONSTANT XOn                : IN    BOOLEAN             := TRUE;
    CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
    CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
   CONSTANT NegPreemptOn      : IN    BOOLEAN            := FALSE;  --IR225 3/14/98
   CONSTANT IgnoreDefaultDelay : IN    BOOLEAN           := FALSE   --IR248 3/14/98
);
PROCEDURE VitalPathDelay01 (
    SIGNAL   OutSignal          : OUT   std_logic;
    VARIABLE GlitchData         : INOUT VitalGlitchDataType;
    CONSTANT OutSignalName      : IN    string;
    CONSTANT OutTemp            : IN    std_logic;
    CONSTANT Paths              : IN    VitalPathArray01Type;
    CONSTANT DefaultDelay       : IN    VitalDelayType01    := VitalZeroDelay01;
    CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
    CONSTANT XOn                : IN    BOOLEAN             := TRUE;
    CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
    CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
   CONSTANT NegPreemptOn      : IN    BOOLEAN            := FALSE;  --IR225 3/14/98
   CONSTANT IgnoreDefaultDelay : IN    BOOLEAN           := FALSE;  --IR248 3/14/98
    CONSTANT RejectFastPath     : IN    BOOLEAN             := FALSE   --IR250
);
PROCEDURE VitalPathDelay01Z (
    SIGNAL   OutSignal          : OUT   std_logic;
    VARIABLE GlitchData         : INOUT VitalGlitchDataType;
    CONSTANT OutSignalName      : IN    string;
    CONSTANT OutTemp            : IN    std_logic;
    CONSTANT Paths              : IN    VitalPathArray01ZType;
    CONSTANT DefaultDelay       : IN    VitalDelayType01Z   := VitalZeroDelay01Z;
    CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
    CONSTANT XOn                : IN    BOOLEAN             := TRUE;
    CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
    CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
   CONSTANT OutputMap         : IN   VitalOutputMapType  := VitalDefaultOutputMap;
   CONSTANT NegPreemptOn      : IN    BOOLEAN            := FALSE;  --IR225 3/14/98
   CONSTANT IgnoreDefaultDelay : IN    BOOLEAN           := FALSE;  --IR248 3/14/98
    CONSTANT RejectFastPath     : IN    BOOLEAN             := FALSE   --IR250
);

-- ------------------------------------------------------------------------
--
-- Function Name:   VitalWireDelay
--
-- Description:     VitalWireDelay is used to delay an input signal.
--                  The delay is selected from the input parameter passed.
--                  The function is useful for back annotation of actual
--                  net delays.
```

Copyright © 2001 IEEE. All rights reserved.

```
--
--                  The function is overloaded to permit passing a delay
--                  value for twire for VitalDelayType, VitalDelayType01
--                  and VitalDelayType01Z.  twire is a generic which can
--                  be back annotated and must be constructed to follow
--                  the SDF to generic mapping rules.
--
-- Arguments:
--
--  IN          Type                  Description
--   InSig       std_ulogic            The input signal (port) to be
--                                     delayed.
--   twire       VitalDelayType        The delay value for which the input
--               VitalDelayType01      signal should be delayed.  For Vital-
--               VitalDelayType01Z     DelayType, the value is single value
--                                     passed.  For VitalDelayType01 and
--                                     VitalDelayType01Z, the appropriate
--                                     delay value is selected by VitalCalc-
--                                     Delay.
--
--  INOUT
--   none
--
--  OUT
--   OutSig      std_ulogic            The internal delayed signal
--
--  Returns
--   none
--
-- ---------------------------------------------------------------------------
PROCEDURE VitalWireDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT twire      : IN    VitalDelayType
    );

PROCEDURE VitalWireDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT twire      : IN    VitalDelayType01
    );

PROCEDURE VitalWireDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT twire      : IN    VitalDelayType01Z
    );

-- --------------------------------------------------------------------------
--
-- Function Name:   VitalSignalDelay
--
-- Description:     The VitalSignalDelay procedure is called in a signal
--                  delay block in the architecture to delay the
--                  appropriate test or reference signal in order to
--                  accommodate negative constraint checks.
--
--                  The amount of delay is of type TIME and is a constant.
--
-- Arguments:
--
--  IN                Type            Description
--   InSig             std_ulogic      The signal to be delayed.
--   dly               TIME            The amount of time the signal is
--                                     delayed.
--
--  INOUT
--   none
--
--  OUT
--   OutSig            std_ulogic      The delayed signal
--
--  Returns
--   none
--
-- ---------------------------------------------------------------------------
PROCEDURE VitalSignalDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT dly        : IN   TIME
);
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- ------------------------------------------------------------------------
--
-- Function Name:  VitalSetupHoldCheck
--
-- Description:    The VitalSetupHoldCheck procedure detects a setup or a
--                 hold violation on the input test signal with respect
--                 to the corresponding input reference signal.  The timing
--                 constraints are specified through parameters
--                 representing the high and low values for the setup and
--                 hold values for the setup and hold times.  This
--                 procedure assumes non-negative values for setup and hold
--                 timing constraints.
--
--                 It is assumed that negative timing constraints
--                 are handled by internally delaying the test or
--                 reference signals.  Negative setup times result in
--                 a delayed reference signal.  Negative hold times
--                 result in a delayed test signal.  Furthermore, the
--                 delays and constraints associated with these and
--                 other signals may need to be appropriately
--                 adjusted so that all constraint intervals overlap
--                 the delayed reference signals and all constraint
--                 values (with respect to the delayed signals) are
--                 non-negative.
--
--                 This function is overloaded based on the input
--                 TestSignal.  A vector and scalar form are provided.
--
-- TestSignal XXXXXXXXXXXX____________________________XXXXXXXXXXXXXXXXXXXXXX
--            :
--            :        -->|       error region       |<--
--            :
--             _______________________________
-- RefSignal                                  \______________________________
--            :           |                   |       |
--            :           |                -->|       |<-- thold
--            :        -->|     tsetup        |<--
--
-- Arguments:
--
--  IN                 Type            Description
--   TestSignal         std_ulogic     Value of test signal
--                      std_logic_vector
--   TestSignalName     STRING         Name of test signal
--   TestDelay          TIME           Model's internal delay associated
--                                     with TestSignal
--   RefSignal          std_ulogic     Value of reference signal
--   RefSignalName      STRING         Name of reference signal
--   RefDelay           TIME           Model's internal delay associated
--                                     with RefSignal
--   SetupHigh          TIME           Absolute minimum time duration before
--                                     the transition of RefSignal for which
--                                     transitions of TestSignal are allowed
--                                     to proceed to the "1" state without
--                                     causing a setup violation.
--   SetupLow           TIME           Absolute minimum time duration before
--                                     the transition of RefSignal for which
--                                     transitions of TestSignal are allowed
--                                     to proceed to the "0" state without
--                                     causing a setup violation.
--   HoldHigh           TIME           Absolute minimum time duration after
--                                     the transition of RefSignal for which
--                                     transitions of TestSignal are allowed
--                                     to proceed to the "1" state without
--                                     causing a hold violation.
--   HoldLow            TIME           Absolute minimum time duration after
--                                     the transition of RefSignal for which
--                                     transitions of TestSignal are allowed
--                                     to proceed to the "0" state without
--                                     causing a hold violation.
--   CheckEnabled       BOOLEAN        Check performed if TRUE.
--   RefTransition      VitalEdgeSymbolType
--                                     Reference edge specified. Events on
--                                     the RefSignal which match the edge
--                                     spec. are used as reference edges.
--   HeaderMsg          STRING         String that will accompany any
--                                     assertion messages produced.
--   XOn                BOOLEAN         If TRUE, Violation output parameter
--                                     is set to "X". Otherwise, Violation
--                                     is always set to "0".
```

Copyright © 2001 IEEE. All rights reserved.

```
--   MsgOn              BOOLEAN        If TRUE, set and hold violation
--                                     message will be generated.
--                                     Otherwise, no messages are generated,
--                                     even upon violations.
--   MsgSeverity        SEVERITY_LEVEL Severity level for the assertion.
--   EnableSetupOnTest  BOOLEAN        If FALSE at the time that the
--                                     TestSignal signal changes,
--                                     no setup check will be performed.
--   EnableSetupOnRef   BOOLEAN        If FALSE at the time that the
--                                     RefSignal signal changes,
--                                     no setup check will be performed.
--   EnableHoldOnRef    BOOLEAN        If FALSE at the time that the
--                                     RefSignal signal changes,
--                                     no hold check will be performed.
--   EnableHoldOnTest   BOOLEAN        If FALSE at the time that the
--                                     TestSignal signal changes,
--                                     no hold check will be performed.
--
--  INOUT
--   TimingData         VitalTimingDataType
--                                     VitalSetupHoldCheck information
--                                     storage area.  This is used
--                                     internally to detect reference edges
--                                     and record the time of the last edge.
--
--  OUT
--   Violation   X01                   This is the violation flag returned.
--
--  Returns
--   none
--
-- ------------------------------------------------------------------------
PROCEDURE VitalSetupHoldCheck (
        VARIABLE Violation     : OUT    X01;
        VARIABLE TimingData    : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal    : IN     std_ulogic;
        CONSTANT TestSignalName: IN     STRING := "";
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        SIGNAL   RefSignal     : IN     std_ulogic;
        CONSTANT RefSignalName : IN     STRING := "";
        CONSTANT RefDelay      : IN     TIME := 0 ns;
        CONSTANT SetupHigh     : IN     TIME := 0 ns;
        CONSTANT SetupLow      : IN     TIME := 0 ns;
        CONSTANT HoldHigh      : IN     TIME := 0 ns;
        CONSTANT HoldLow       : IN     TIME := 0 ns;
        CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg     : IN     STRING := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT EnableSetupOnTest : IN BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableSetupOnRef  : IN BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableHoldOnRef   : IN BOOLEAN := TRUE;    --IR252 3/23/98
    CONSTANT EnableHoldOnTest  : IN BOOLEAN := TRUE    --IR252 3/23/98
  );

PROCEDURE VitalSetupHoldCheck (
        VARIABLE Violation     : OUT    X01;
        VARIABLE TimingData    : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal    : IN     std_logic_vector;
        CONSTANT TestSignalName: IN     STRING := "";
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        SIGNAL   RefSignal     : IN     std_ulogic;
        CONSTANT RefSignalName : IN     STRING := "";
        CONSTANT RefDelay      : IN     TIME := 0 ns;
        CONSTANT SetupHigh     : IN     TIME := 0 ns;
        CONSTANT SetupLow      : IN     TIME := 0 ns;
        CONSTANT HoldHigh      : IN     TIME := 0 ns;
        CONSTANT HoldLow       : IN     TIME := 0 ns;
        CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg     : IN     STRING := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT EnableSetupOnTest : IN BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableSetupOnRef  : IN BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableHoldOnRef   : IN BOOLEAN := TRUE;    --IR252 3/23/98
    CONSTANT EnableHoldOnTest  : IN BOOLEAN := TRUE    --IR252 3/23/98
);
```

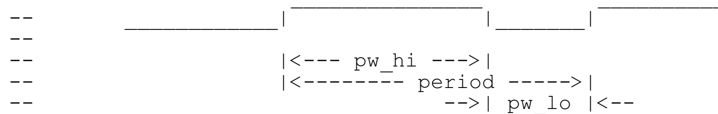
 Copyright © 2001 IEEE. All rights reserved.

```
-- ---------------------------------------------------------------------
--
-- Function Name:   VitalRecoveryRemovalCheck
--
-- Description:     The VitalRecoveryRemovalCheck detects the presence of
--                  a recovery or removal violation on the input test
--                  signal with respect to the corresponding input reference
--                  signal.  It assumes non-negative values of setup and
--                  hold timing constraints.  The timing constraint is
--                  specified through parameters representing the recovery
--                  and removal times associated with a reference edge of
--                  the reference signal.  A flag indicates whether a test
--                  signal is asserted when it is high or when it is low.
--
--                  It is assumed that negative timing constraints
--                  are handled by internally delaying the test or
--                  reference signals.  Negative recovery times result in
--                  a delayed reference signal.  Negative removal times
--                  result in a delayed test signal.  Furthermore, the
--                  delays and constraints associated with these and
--                  other signals may need to be appropriately
--                  adjusted so that all constraint intervals overlap
--                  the delayed reference signals and all constraint
--                  values (with respect to the delayed signals) are
--                  non-negative.
--
-- Arguments:
--
--  IN                Type              Description
--   TestSignal       std_ulogic        Value of TestSignal.  The routine is
--   TestSignalName   STRING            Name of TestSignal
--   TestDelay        TIME              Model internal delay associated with
--                                      the TestSignal
--   RefSignal        std_ulogic        Value of RefSignal
--   RefSignalName    STRING            Name of RefSignal
--   RefDelay         TIME              Model internal delay associated with
--                                      the RefSignal
--   Recovery         TIME              A change to an unasserted value on
--                                      the asynchronous TestSignal must
--                                      precede reference edge (on RefSignal)
--                                      by at least this time.
--   Removal          TIME              An asserted condition must be present
--                                      on the asynchronous TestSignal for at
--                                      least the removal time following a
--                                      reference edge on RefSignal.
--   ActiveLow        BOOLEAN           A flag which indicates if TestSignal
--                                      is asserted when it is low - “0.”
--                                      FALSE indicate that TestSignal is
--                                      asserted when it has a value “1.”
--   CheckEnabled     BOOLEAN           The check in enabled when the value
--                                      is TRUE, otherwise the constraints
--                                      are not checked.
--   RefTransition    VitalEdgeSymbolType
--                                      Reference edge specifier.  Events on
--                                      RefSignal will match the edge
--                                      specified.
--   HeaderMsg         STRING           A header message that will accompany
--                                      any assertion message.
--   XOn               BOOLEAN          When TRUE, the output Violation is
--                                      set to “X.”  When FALSE, it is always
--                                      “0.”
--   MsgOn             BOOLEAN          When TRUE, violation messages are
--                                      output.  When FALSE, no messages are
--                                      generated.
--   MsgSeverity       SEVERITY_LEVEL  Severity level of the asserted
--                                      message.
--   EnableRecOnTest   BOOLEAN          If FALSE at the time that the
--                                      TestSignal signal changes,
--                                      no recovery check will be performed.
--   EnableRecOnRef    BOOLEAN          If FALSE at the time that the
--                                      RefSignal signal changes,
--                                      no recovery check will be performed.
--   EnableRemOnRef    BOOLEAN          If FALSE at the time that the
--                                      RefSignal signal changes,
--                                      no removal check will be performed.
--   EnableRemOnTest   BOOLEAN          If FALSE at the time that the
--                                      TestSignal signal changes,
--                                      no removal check will be performed.
--
```

Copyright © 2001 IEEE. All rights reserved.

```
--   INOUT
--    TimingData       VitalTimingDataType
--                                    VitalRecoveryRemovalCheck information
--                                    storage area.  This is used
--                                    internally to detect reference edges
--                                    and record the time of the last edge.
--   OUT
--    Violation        X01            This is the violation flag returned.
--
--   Returns
--           none
--
-- -------------------------------------------------------------------------
PROCEDURE VitalRecoveryRemovalCheck (
        VARIABLE Violation      : OUT    X01;
        VARIABLE TimingData     : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal     : IN     std_ulogic;
        CONSTANT TestSignalName: IN      STRING := “”;
        CONSTANT TestDelay      : IN     TIME := 0 ns;
        SIGNAL   RefSignal      : IN     std_ulogic;
        CONSTANT RefSignalName  : IN     STRING := “”;
        CONSTANT RefDelay       : IN     TIME := 0 ns;
        CONSTANT Recovery       : IN     TIME := 0 ns;
        CONSTANT Removal        : IN     TIME := 0 ns;
        CONSTANT ActiveLow      : IN     BOOLEAN := TRUE;
        CONSTANT CheckEnabled   : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition  : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg      : IN     STRING := “ “;
        CONSTANT XOn            : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn          : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity    : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT EnableRecOnTest : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableRecOnRef  : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableRemOnRef  : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableRemOnTest : IN   BOOLEAN := TRUE--IR252 3/23/98
);

-- -------------------------------------------------------------------------
--
-- Function Name:  VitalPeriodPulseCheck
--
-- Description:    VitalPeriodPulseCheck checks for minimum
--                 periodicity and pulse width for “1” and “0” values of
--                 the input test signal.  The timing constraint is
--                 specified through parameters representing the minimal
--                 period between successive rising and falling edges of
--                 the input test signal and the minimum pulse widths
--                 associated with high and low values.
--
--                 VitalPeriodCheck’s accepts rising and falling edges
--                 from 1 and 0 as well as transitions to and from ‘X.’
--
```

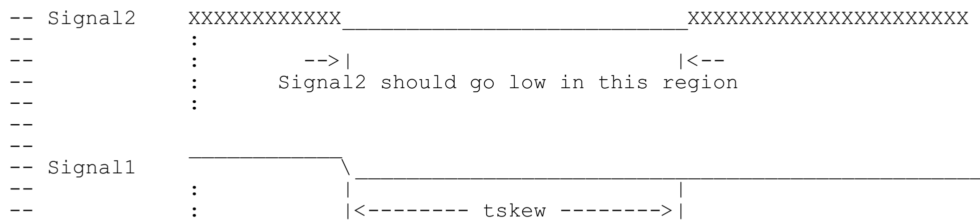


```
--                  _______________         __________
--       ____________|               |_______|
--
--                    |<--- pw_hi --->|
--                    |<-------- period ----->|
--                                -->| pw_lo |<--
--
-- Arguments:
--  IN                 Type           Description
--    TestSignal        std_ulogic     Value of test signal
--    TestSignalName    STRING         Name of the test signal
--    TestDelay         TIME           Model’s internal delay associated
--                                     with TestSignal
--    Period            TIME           Minimum period allowed between
--                                     consecutive rising (‘P’) or
--                                     falling (‘F’) transitions.
--    PulseWidthHigh    TIME           Minimum time allowed for a high
--                                     pulse (‘1’ or ‘H’)
--    PulseWidthLow     TIME           Minimum time allowed for a low
--                                     pulse (‘0’ or ‘L’)
--    CheckEnabled      BOOLEAN        Check performed if TRUE.
--    HeaderMsg         STRING         String that will accompany any
--                                     assertion messages produced.
--    XOn               BOOLEAN        If TRUE, Violation output parameter
--                                     is set to “X”. Otherwise, Violation
--                                     is always set to “0”.
--                                     XOnChecks is a global that allows for
--                                     only timing checks to be turned on.
--    MsgOn             BOOLEAN        If TRUE, period/pulse violation
```

 Copyright © 2001 IEEE. All rights reserved.

```
--                                     message will be generated.
--                                     Otherwise, no messages are generated,
--                                     even though a violation is detected.
--                                     MsgOnChecks allows for only timing
--                                     check messages to be turned on.
--    MsgSeverity       SEVERITY_LEVEL Severity level for the assertion.
--
--   INOUT
--    PeriodData        VitalPeriodDataType
--                                     VitalPeriodPulseCheck information
--                                     storage area.  This is used
--                                     internally to detect reference edges
--                                     and record the pulse and period
--                                     times.
--   OUT
--    Violation         X01            This is the violation flag returned.
--
--   Returns
--    none
--
-- -------------------------------------------------------------------------
PROCEDURE VitalPeriodPulseCheck  (
        VARIABLE Violation      : OUT    X01;
        VARIABLE PeriodData     : INOUT  VitalPeriodDataType;
        SIGNAL   TestSignal     : IN     std_ulogic;
        CONSTANT TestSignalName : IN     STRING := "";
        CONSTANT TestDelay      : IN     TIME := 0 ns;
        CONSTANT Period         : IN     TIME := 0 ns;
        CONSTANT PulseWidthHigh : IN     TIME := 0 ns;
        CONSTANT PulseWidthLow  : IN     TIME := 0 ns;
        CONSTANT CheckEnabled   : IN     BOOLEAN := TRUE;
        CONSTANT HeaderMsg      : IN     STRING := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING
    );

-- -------------------------------------------------------------------------
--
-- Function Name:  VitalInPhaseSkewCheck
--
-- Description:    The VitalInPhaseSkewCheck procedure detects an in-phase
--                 skew violation between input signals Signal1 and Signal2.
--                 This is a timer based skew check in which a
--                 violation is detected if Signal1 and Signal2 are in
--                 different logic states longer than the specified skew
--                 interval.
--
--                 The timing constraints are specified through parameters
--                 representing the skew values for the different states
--                 of Signal1 and Signal2.
--
--
```




```
--
-- Arguments:
--
--  IN                 Type           Description
--   Signal1            std_ulogic     Value of first signal
--   Signal1Name        STRING         Name of first signal
--   Signal1Delay       TIME           Model's internal delay associated
--                                     with Signal1
--   Signal2            std_ulogic     Value of second signal
--   Signal2Name        STRING         Name of second signal
--   Signal2Delay       TIME           Model's internal delay associated
--                                     with Signal2
--   SkewS1S2RiseRise   TIME           Absolute maximum time duration for
--                                     which Signal2 can remain at "0"
--                                     after Signal1 goes to the "1" state,
--                                     without causing a skew violation.
--   SkewS2S1RiseRise   TIME           Absolute maximum time duration for
--                                     which Signal1 can remain at "0"
```

Copyright © 2001 IEEE. All rights reserved.

```
--                                      after Signal2 goes to the “1” state,
--                                      without causing a skew violation.
--   SkewS1S2FallFall   TIME            Absolute maximum time duration for
--                                      which Signal2 can remain at “1”
--                                      after Signal1 goes to the “0” state,
--                                      without causing a skew violation.
--   SkewS2S1FallFall   TIME            Absolute maximum time duration for
--                                      which Signal1 can remain at “1”
--                                      after Signal2 goes to the “0” state,
--                                      without causing a skew violation.
--   CheckEnabled       BOOLEAN         Check performed if TRUE.
--   HeaderMsg          STRING          String that will accompany any
--                                      assertion messages produced.
--   XOn                BOOLEAN         If TRUE, Violation output parameter
--                                      is set to “X”. Otherwise, Violation
--                                      is always set to “0.”
--   MsgOn              BOOLEAN         If TRUE, skew timing violation
--                                      messages will be generated.
--                                      Otherwise, no messages are generated,
--                                      even upon violations.
--   MsgSeverity        SEVERITY_LEVEL Severity level for the assertion.
--
--  INOUT
--   SkewData           VitalSkewDataType
--                                      VitalInPhaseSkewCheck information
--                                      storage area.  This is used
--                                      internally to detect signal edges
--                                      and record the time of the last edge.
--
--
--   Trigger            std_ulogic      This signal is used to trigger the
--                                      process in which the timing check
--                                      occurs upon expiry of the skew
--                                      interval.
--
--  OUT
--   Violation   X01                    This is the violation flag returned.
--
--  Returns
--   none
--
-- -------------------------------------------------------------------------

PROCEDURE VitalInPhaseSkewCheck (
    VARIABLE Violation         : OUT    X01;
    VARIABLE SkewData          : INOUT  VitalSkewDataType;
    SIGNAL   Signal1           : IN     std_ulogic;
    CONSTANT Signal1Name       : IN     STRING := “”;
    CONSTANT Signal1Delay      : IN     TIME := 0 ns;
    SIGNAL   Signal2           : IN     std_ulogic;
    CONSTANT Signal2Name       : IN     STRING := “”;
    CONSTANT Signal2Delay      : IN     TIME := 0 ns;
    CONSTANT SkewS1S2RiseRise  : IN     TIME := TIME’HIGH;
    CONSTANT SkewS2S1RiseRise  : IN     TIME := TIME’HIGH;
    CONSTANT SkewS1S2FallFall  : IN     TIME := TIME’HIGH;
    CONSTANT SkewS2S1FallFall  : IN     TIME := TIME’HIGH;
    CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
    CONSTANT XOn               : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT HeaderMsg         : IN     STRING  := “”;
    SIGNAL   Trigger           : INOUT  std_ulogic
);


-- -------------------------------------------------------------------------
--
-- Function Name:  VitalOutPhaseSkewCheck
--
-- Description:    The VitalOutPhaseSkewCheck procedure detects an
--                 out-of-phase skew violation between input signals Signal1
--                 and Signal2. This is a timer based skew check in
--                 which a violation is detected if Signal1 and Signal2 are
--                 in the same logic state longer than the specified skew
--                 interval.
--
--                 The timing constraints are specified through parameters
--                 representing the skew values for the different states
--                 of Signal1 and Signal2.
--
--
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- Signal2    XXXXXXXXXXXX___________________________XXXXXXXXXXXXXXXXXXXXXX
--            :
--            :        -->|                         |<--
--            :     Signal2 should go high in this region
--            :
--
--            ____________
-- Signal1                \_________________________________________________
--            :           |                         |
--            :           |<-------- tskew -------->|
--
-- Arguments:
--
--  IN                  Type           Description
--   Signal1             std_ulogic     Value of first signal
--   Signal1Name         STRING         Name of first signal
--   Signal1Delay        TIME           Model's internal delay associated
--                                      with Signal1
--   Signal2             std_ulogic     Value of second signal
--   Signal2Name         STRING         Name of second signal
--   Signal2Delay        TIME           Model's internal delay associated
--                                      with Signal2
--   SkewS1S2RiseFall    TIME           Absolute maximum time duration for
--                                      which Signal2 can remain at "1"
--                                      after Signal1 goes to the "1" state,
--                                      without causing a skew violation.
--   SkewS2S1RiseFall    TIME           Absolute maximum time duration for
--                                      which Signal1 can remain at "1"
--                                      after Signal2 goes to the "1" state,
--                                      without causing a skew violation.
--   SkewS1S2FallRise    TIME           Absolute maximum time duration for
--                                      which Signal2 can remain at "0"
--                                      after Signal1 goes to the "0" state,
--                                      without causing a skew violation.
--   SkewS2S1FallRise    TIME           Absolute maximum time duration for
--                                      which Signal1 can remain at "0"
--                                      after Signal2 goes to the "0" state,
--                                      without causing a skew violation.
--   CheckEnabled        BOOLEAN        Check performed if TRUE.
--   HeaderMsg           STRING         String that will accompany any
--                                      assertion messages produced.
--   XOn                 BOOLEAN        If TRUE, Violation output parameter
--                                      is set to "X". Otherwise, Violation
--                                      is always set to "0."
--   MsgOn               BOOLEAN        If TRUE, skew timing violation
--                                      messages will be generated.
--                                      Otherwise, no messages are generated,
--                                      even upon violations.
--   MsgSeverity         SEVERITY_LEVEL Severity level for the assertion.
--
--  INOUT
--   SkewData            VitalSkewDataType
--                                      VitalInPhaseSkewCheck information
--                                      storage area.  This is used
--                                      internally to detect signal edges
--                                      and record the time of the last edge.
--
--   Trigger             std_ulogic     This signal is used to trigger the
--                                      process in which the timing check
--                                      occurs upon expiry of the skew
--                                      interval.
--
--  OUT
--   Violation   X01                    This is the violation flag returned.
--
--  Returns
--   none
--
-- -------------------------------------------------------------------------
PROCEDURE VitalOutPhaseSkewCheck (
    VARIABLE Violation         : OUT    X01;
    VARIABLE SkewData          : INOUT  VitalSkewDataType;
    SIGNAL   Signal1           : IN     std_ulogic;
    CONSTANT Signal1Name       : IN     STRING := "";
    CONSTANT Signal1Delay      : IN     TIME := 0 ns;
    SIGNAL   Signal2           : IN     std_ulogic;
    CONSTANT Signal2Name       : IN     STRING := "";
    CONSTANT Signal2Delay      : IN     TIME := 0 ns;
    CONSTANT SkewS1S2RiseFall  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS2S1RiseFall  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS1S2FallRise  : IN     TIME := TIME'HIGH;
```

Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT SkewS2S1FallRise  : IN     TIME := TIME'HIGH;
        CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
        CONSTANT XOn               : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT HeaderMsg         : IN     STRING  := "";
        SIGNAL   Trigger           : INOUT  std_ulogic
    );

END VITAL_Timing;
```

13.2 VITAL_Timing package body

```
-------------------------------------------------------------------------------
-- Title        : Standard VITAL TIMING Package
--              : $Revision: 1.5 $
-- Library      : VITAL
--              :
-- Developers   : IEEE DASC Timing Working Group (TWG), PAR 1076.4
--              :
-- Purpose      : This packages defines standard types, attributes, constants,
--              : functions and procedures for use in developing ASIC models.
--              : This file contains the Package Body.
-- ----------------------------------------------------------------------------
--
-- ----------------------------------------------------------------------------
-- Modification History :
-- ----------------------------------------------------------------------------
-- Version No:|Auth:| Mod.Date:| Changes Made:
--   v95.0 A  |     | 06/08/95 | Initial ballot draft 1995
--   v95.1    |     | 08/31/95 | #IR203 - Timing violations at time 0
--                               #IR204 - Output mapping prior to glitch detection
--   v98.0    |TAG  | 03/27/98 | Initial ballot draft 1998
--                             | #IR225 - Negative Premptive Glitch
--                                      **Code_effected=ReportGlitch,VitalGlitch,
--                                  VitalPathDelay,VitalPathDelay01,
--                                      VitalPathDelay01z.
--                               #IR105 - Skew timing check needed
--                                      **Code_effected=NONE, New code added!!
--                              #IR245,IR246,IR251 ITC code to fix false boundry cases
--                                 **Code_effected=InternalTimingCheck.
--                          #IR248 - Allows VPD to use a default timing delay
--                                      **Code_effected=VitalPathDelay,
--                                        VitalPathDelay01,VitalPathDelay01z,
--                                        VitalSelectPathDelay,VitalSelectPathDelay01,
--                                        VitalSelectPathDelay01z.
--                          #IR250 - Corrects fastpath condition in VPD
--                                      **Code_effected=VitalPathDelay01,
--                                        VitalPathDelay01z,
--                               #IR252 - Corrects cancelled timing check call if
--                                        condition expires.
--                                        **Code_effected=VitalSetupHoldCheck,
--                                        VitalRecoveryRemovalCheck.
--   v98.1    | jdc | 03/25/99 | Changed UseDefaultDelay to IgnoreDefaultDelay
--                               and set default to FALSE in VitalPathDelay()
--
-- ----------------------------------------------------------------------------

LIBRARY STD;
USE STD.TEXTIO.ALL;

PACKAGE BODY VITAL_Timing IS

    -- ---------------------------------------------------------------------
    -- Package Local Declarations
    -- ---------------------------------------------------------------------

    TYPE CheckType IS ( SetupCheck, HoldCheck, RecoveryCheck, RemovalCheck,
                        PulseWidCheck, PeriodCheck );

    TYPE CheckInfoType IS RECORD
            Violation : BOOLEAN;
            CheckKind : CheckType;
            ObsTime   : TIME;
            ExpTime   : TIME;
            DetTime   : TIME;
            State     : X01;
```

 Copyright © 2001 IEEE. All rights reserved.

```
END RECORD;

TYPE LogicCvtTableType IS ARRAY (std_ulogic) OF CHARACTER;
TYPE HiLoStrType IS ARRAY (std_ulogic RANGE 'X' TO '1') OF STRING(1 TO 4);

CONSTANT LogicCvtTable : LogicCvtTableType
                 := ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
CONSTANT HiLoStr     : HiLoStrType := ("  X ", " Low", "High" );

TYPE EdgeSymbolMatchType IS ARRAY (X01,X01,VitalEdgeSymbolType) OF BOOLEAN;
-- last value, present value, edge symbol
CONSTANT EdgeSymbolMatch : EdgeSymbolMatchType := (
  'X'=>('X'=>(                                  OTHERS => FALSE),
        '0'=>('N'|'F'|'v'|'E'|'D'|'*' => TRUE, OTHERS => FALSE ),
        '1'=>('P'|'R'|'^'|'E'|'A'|'*' => TRUE, OTHERS => FALSE ) ),
  '0'=>('X'=>(    'r'|'p'|'R'|'A'|'*' => TRUE, OTHERS => FALSE ),
        '0'=>(                                  OTHERS => FALSE ),
        '1'=>(    '/'|'P'|'p'|'R'|'*' => TRUE, OTHERS => FALSE ) ),
  '1'=>('X'=>(    'f'|'n'|'F'|'D'|'*' => TRUE, OTHERS => FALSE ),
        '0'=>(    '\'|'N'|'n'|'F'|'*' => TRUE, OTHERS => FALSE ),
        '1'=>(                                  OTHERS => FALSE ) ) );



---------------------------------------------------------------------------
-- Tables used to implement 'posedge' and 'negedge' in path delays
-- These are new tables for Skewcheck routines.  IR105
---------------------------------------------------------------------------

TYPE EdgeRable IS ARRAY(std_ulogic, std_ulogic) OF boolean;

CONSTANT Posedge : EdgeRable := (
-- ------------------------------------------------------------------------
-- | U      X      0      1      Z      W      L      H      -
-- ------------------------------------------------------------------------
   ( FALSE, FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE ), -- U
   ( FALSE, FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE ), -- X
   ( TRUE , TRUE , FALSE, TRUE , TRUE , TRUE , FALSE, TRUE , TRUE  ), -- 0
   ( FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE ), -- 1
   ( FALSE, FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE ), -- Z
   ( FALSE, FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE ), -- W
   ( TRUE , TRUE , FALSE, TRUE , TRUE , TRUE , FALSE, TRUE , TRUE  ), -- L
   ( FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE ), -- H
   ( FALSE, FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE )  -- -

); --IR105


CONSTANT Negedge : EdgeRable := (
-- -----------------------------------------------------------------------
-- | U      X      0      1      Z      W      L      H      -
-- -----------------------------------------------------------------------
   ( FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE, FALSE ), -- U
   ( FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE, FALSE ), -- X
   ( FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE ), -- 0
   ( TRUE , TRUE , TRUE , FALSE, TRUE , TRUE , TRUE , FALSE, TRUE  ), -- 1
   ( FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE, FALSE ), -- Z
   ( FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE, FALSE ), -- W
   ( FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE ), -- L
   ( TRUE , TRUE , TRUE , FALSE, TRUE , TRUE , TRUE , FALSE, TRUE  ), -- H
   ( FALSE, FALSE, TRUE , FALSE, FALSE, FALSE, TRUE , FALSE, FALSE ) --  -

);  --IR105

TYPE SkewType IS (Inphase, Outphase);  --IR105

CONSTANT noTrigger : TIME := -1 ns;   --IR105
---------------------------------------------------------------------------
--    End of Skew (IR105 additions)
---------------------------------------------------------------------------


---------------------------------------------------------------------------
---------------------------------------------------------------------------
-- Misc Utilities Local Utilities
---------------------------------------------------------------------------
-----------------------------------------------------------------------
FUNCTION Minimum ( CONSTANT t1,t2 : IN TIME ) RETURN TIME IS
BEGIN
    IF ( t1 < t2 ) THEN RETURN (t1); ELSE RETURN (t2); END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
END Minimum;
------------------------------------------------------------------------
FUNCTION Maximum ( CONSTANT t1,t2 : IN TIME ) RETURN TIME IS
BEGIN
    IF ( t1 > t2 ) THEN RETURN (t1); ELSE RETURN (t2); END IF;
END Maximum;

---------------------------------------------------------------------
-- Error Message Types and Tables
---------------------------------------------------------------------
TYPE VitalErrorType IS (
    ErrVctLng  ,
    ErrNoPath  ,
    ErrNegPath ,
    ErrNegDel
);

TYPE VitalErrorSeverityType IS ARRAY (VitalErrorType) OF SEVERITY_LEVEL;
CONSTANT VitalErrorSeverity : VitalErrorSeverityType := (
    ErrVctLng    => ERROR,
    ErrNoPath    => WARNING,
    ErrNegPath   => WARNING,
    ErrNegDel    => WARNING
);

CONSTANT MsgNoPath : STRING :=
  "No Delay Path Condition TRUE.  0-delay used. Output signal is: ";
CONSTANT MsgNegPath : STRING :=
  "Path Delay less than time since input. 0 delay used. Output signal is: ";
CONSTANT MsgNegDel : STRING :=
  "Negative delay. New output value not scheduled. Output signal is: ";
CONSTANT MsgVctLng : STRING :=
  "Vector (array) lengths not equal. ";

CONSTANT MsgUnknown : STRING :=
  "Unknown error message.";

FUNCTION VitalMessage (
        CONSTANT ErrorId : IN VitalErrorType
      ) RETURN STRING IS
BEGIN
    CASE ErrorId IS
        WHEN ErrVctLng    => RETURN MsgVctLng;
        WHEN ErrNoPath    => RETURN MsgNoPath;
        WHEN ErrNegPath   => RETURN MsgNegPath;
        WHEN ErrNegDel    => RETURN MsgNegDel;
        WHEN OTHERS       => RETURN MsgUnknown;
    END CASE;
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId)
      SEVERITY VitalErrorSeverity(ErrorId);
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType;
        CONSTANT Info    : IN STRING
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId) & Info
      SEVERITY VitalErrorSeverity(ErrorId);
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType;
        CONSTANT Info    : IN CHARACTER
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId) & Info
      SEVERITY VitalErrorSeverity(ErrorId);
END;
```

 Copyright © 2001 IEEE. All rights reserved.

```
---------------------------------------------------------------------------
-- Time Delay Assignment Subprograms
---------------------------------------------------------------------------
FUNCTION VitalExtendToFillDelay (
        CONSTANT Delay : IN VitalDelayType
      ) RETURN VitalDelayType01Z IS
BEGIN
    RETURN (OTHERS => Delay);
END VitalExtendToFillDelay;

FUNCTION VitalExtendToFillDelay (
        CONSTANT Delay : IN VitalDelayType01
      ) RETURN VitalDelayType01Z IS
    VARIABLE Delay01Z  : VitalDelayType01Z;
BEGIN
    Delay01Z(tr01) := Delay(tr01);
    Delay01Z(tr0z) := Delay(tr01);
    Delay01Z(trz1) := Delay(tr01);
    Delay01Z(tr10) := Delay(tr10);
    Delay01Z(tr1z) := Delay(tr10);
    Delay01Z(trz0) := Delay(tr10);
    RETURN (Delay01Z);
END VitalExtendToFillDelay;

FUNCTION VitalExtendToFillDelay (
        CONSTANT Delay : IN VitalDelayType01Z
      ) RETURN VitalDelayType01Z IS
BEGIN
    RETURN Delay;
END VitalExtendToFillDelay;

---------------------------------------------------------------------------
FUNCTION VitalCalcDelay (
        CONSTANT NewVal : IN std_ulogic   := 'X';
        CONSTANT OldVal : IN std_ulogic   := 'X';
        CONSTANT Delay  : IN VitalDelayType
      ) RETURN TIME IS
BEGIN
      RETURN delay;
END VitalCalcDelay;

FUNCTION VitalCalcDelay (
        CONSTANT NewVal : IN std_ulogic   := 'X';
        CONSTANT OldVal : IN std_ulogic   := 'X';
        CONSTANT Delay  : IN VitalDelayType01
      ) RETURN TIME IS
    VARIABLE Result : TIME;
BEGIN
    CASE Newval IS
      WHEN '0' | 'L' => Result := Delay(tr10);
      WHEN '1' | 'H' => Result := Delay(tr01);
      WHEN 'Z' =>
        CASE Oldval IS
          WHEN '0' | 'L' => Result := Delay(tr01);
          WHEN '1' | 'H' => Result := Delay(tr10);
          WHEN OTHERS    => Result := MAXIMUM(Delay(tr10), Delay(tr01));
        END CASE;
      WHEN OTHERS =>
        CASE Oldval IS
          WHEN '0' | 'L' => Result := Delay(tr01);
          WHEN '1' | 'H' => Result := Delay(tr10);
          WHEN 'Z'       => Result := MINIMUM(Delay(tr10), Delay(tr01));
          WHEN OTHERS    => Result := MAXIMUM(Delay(tr10), Delay(tr01));
        END CASE;
    END CASE;
    RETURN Result;
END VitalCalcDelay;

FUNCTION VitalCalcDelay (
        CONSTANT NewVal : IN std_ulogic   := 'X';
        CONSTANT OldVal : IN std_ulogic   := 'X';
        CONSTANT Delay  : IN VitalDelayType01Z
      ) RETURN TIME IS
    VARIABLE Result : TIME;
BEGIN
    CASE Oldval IS
      WHEN '0' | 'L' =>
        CASE Newval IS
          WHEN '0' | 'L' => Result := Delay(tr10);
          WHEN '1' | 'H' => Result := Delay(tr01);
```

Copyright © 2001 IEEE. All rights reserved.

```
              WHEN 'Z'        => Result := Delay(tr0z);
              WHEN OTHERS     => Result := MINIMUM(Delay(tr01), Delay(tr0z));
            END CASE;
          WHEN '1' | 'H' =>
            CASE Newval IS
              WHEN '0' | 'L' => Result := Delay(tr10);
              WHEN '1' | 'H' => Result := Delay(tr01);
              WHEN 'Z'        => Result := Delay(tr1z);
              WHEN OTHERS     => Result := MINIMUM(Delay(tr10), Delay(tr1z));
            END CASE;
          WHEN 'Z' =>
            CASE Newval IS
              WHEN '0' | 'L' => Result := Delay(trz0);
              WHEN '1' | 'H' => Result := Delay(trz1);
              WHEN 'Z'        => Result := MAXIMUM (Delay(tr0z), Delay(tr1z));
              WHEN OTHERS     => Result := MINIMUM (Delay(trz1), Delay(trz0));
            END CASE;
          WHEN 'U' | 'X' | 'W' | '-'  =>
            CASE Newval IS
              WHEN '0' | 'L' => Result := MAXIMUM(Delay(tr10), Delay(trz0));
              WHEN '1' | 'H' => Result := MAXIMUM(Delay(tr01), Delay(trz1));
              WHEN 'Z'        => Result := MAXIMUM(Delay(tr1z), Delay(tr0z));
              WHEN OTHERS     => Result := MAXIMUM(Delay(tr10), Delay(tr01));
            END CASE;
      END CASE;
      RETURN Result;
  END VitalCalcDelay;

  ---------------------------------------------------------------------------
  --
  -- VitalSelectPathDelay returns the path delay selected by the Paths array.
  -- If no paths are selected, it returns either the appropriate default
  -- delay or TIME'HIGH, depending upon the value of IgnoreDefaultDelay.
  --

  FUNCTION VitalSelectPathDelay (
          CONSTANT NewValue           : IN  std_logic;
          CONSTANT OldValue           : IN  std_logic;
          CONSTANT OutSignalName      : IN  string;
          CONSTANT Paths              : IN  VitalPathArrayType;
          CONSTANT DefaultDelay       : IN  VitalDelayType;
          CONSTANT IgnoreDefaultDelay : IN  BOOLEAN
        ) RETURN TIME IS

      VARIABLE TmpDelay  : TIME;
      VARIABLE InputAge  : TIME := TIME'HIGH;
      VARIABLE PropDelay : TIME := TIME'HIGH;
  BEGIN
      -- for each delay path
      FOR i IN Paths'RANGE LOOP
          -- ignore the delay path if it is not enabled
          NEXT WHEN NOT Paths(i).PathCondition;
          -- ignore the delay path if a more recent input event has been seen
          NEXT WHEN Paths(i).InputChangeTime > InputAge;

          -- This is the most recent input change (so far)
          -- Get the transition dependent delay
          TmpDelay := VitalCalcDelay(NewValue, OldValue, Paths(i).PathDelay);

          -- If other inputs changed at the same time,
          -- then use the minimum of their propagation delays,
          -- else use the propagation delay from this input.
          IF Paths(i).InputChangeTime < InputAge THEN
              PropDelay := TmpDelay;
          ELSE -- Simultaneous inputs change
              IF TmpDelay < PropDelay THEN PropDelay := TmpDelay; END IF;
          end if;

          InputAge := Paths(i).InputChangeTime;
      END LOOP;

      -- If there were no paths (with an enabled condition),
      -- use the default delay, if so indicated, otherwise return TIME'HIGH
      IF (PropDelay = TIME'HIGH) THEN
IF (IgnoreDefaultDelay) THEN
          PropDelay := VitalCalcDelay(NewValue, OldValue, DefaultDelay);
END IF;

-- If the time since the most recent selected input event is
-- greater than the propagation delay from that input,
-- then use the default delay (won't happen if no paths are selected)
```

 Copyright © 2001 IEEE. All rights reserved.

```
        ELSIF (InputAge > PropDelay) THEN
            PropDelay := VitalCalcDelay(NewValue, OldValue, DefaultDelay);

        -- Adjust the propagation delay by the time since the
        -- the input event occurred  (Usually 0 ns).
        ELSE
            PropDelay := PropDelay - InputAge;
        END IF;

        RETURN PropDelay;
    END;

    FUNCTION VitalSelectPathDelay (
            CONSTANT NewValue           : IN  std_logic;
            CONSTANT OldValue           : IN  std_logic;
            CONSTANT OutSignalName      : IN  string;
            CONSTANT Paths              : IN  VitalPathArray01Type;
            CONSTANT DefaultDelay       : IN  VitalDelayType01;
            CONSTANT IgnoreDefaultDelay : IN  BOOLEAN
          ) RETURN TIME IS

        VARIABLE TmpDelay  : TIME;
        VARIABLE InputAge  : TIME := TIME'HIGH;
        VARIABLE PropDelay : TIME := TIME'HIGH;
    BEGIN
        -- for each delay path
        FOR i IN Paths'RANGE LOOP
            -- ignore the delay path if it is not enabled
            NEXT WHEN NOT Paths(i).PathCondition;
            -- ignore the delay path if a more recent input event has been seen
            NEXT WHEN Paths(i).InputChangeTime > InputAge;

            -- This is the most recent input change (so far)
            -- Get the transition dependent delay
            TmpDelay := VitalCalcDelay(NewValue, OldValue, Paths(i).PathDelay);

            -- If other inputs changed at the same time,
            -- then use the minimum of their propagation delays,
            -- else use the propagation delay from this input.
            IF Paths(i).InputChangeTime < InputAge THEN
                PropDelay := TmpDelay;
            ELSE -- Simultaneous inputs change
                IF TmpDelay < PropDelay THEN PropDelay := TmpDelay; END IF;
            end if;

            InputAge := Paths(i).InputChangeTime;
        END LOOP;

        -- If there were no paths (with an enabled condition),
        -- use the default delay, if so indicated, otherwise return TIME'HIGH
        IF (PropDelay = TIME'HIGH) THEN
  IF (IgnoreDefaultDelay) THEN
            PropDelay := VitalCalcDelay(NewValue, OldValue, DefaultDelay);
  END IF;

-- If the time since the most recent selected input event is
-- greater than the propagation delay from that input,
-- then use the default delay (won't happen if no paths are selected)
        ELSIF (InputAge > PropDelay) THEN
            PropDelay := VitalCalcDelay(NewValue, OldValue, DefaultDelay);

        -- Adjust the propagation delay by the time since the
        -- the input event occurred  (Usually 0 ns).
        ELSE
            PropDelay := PropDelay - InputAge;
        END IF;

        RETURN PropDelay;
    END;

    FUNCTION VitalSelectPathDelay (
            CONSTANT NewValue           : IN  std_logic;
            CONSTANT OldValue           : IN  std_logic;
            CONSTANT OutSignalName      : IN  string;
            CONSTANT Paths              : IN  VitalPathArray01ZType;
            CONSTANT DefaultDelay       : IN  VitalDelayType01Z;
        CONSTANT IgnoreDefaultDelay : IN  BOOLEAN
          ) RETURN TIME IS

        VARIABLE TmpDelay  : TIME;
        VARIABLE InputAge  : TIME := TIME'HIGH;
```


Copyright © 2001 IEEE. All rights reserved.

```
        VARIABLE PropDelay : TIME := TIME'HIGH;
    BEGIN
        -- for each delay path
        FOR i IN Paths'RANGE LOOP
            -- ignore the delay path if it is not enabled
            NEXT WHEN NOT Paths(i).PathCondition;
            -- ignore the delay path if a more recent input event has been seen
            NEXT WHEN Paths(i).InputChangeTime > InputAge;

            -- This is the most recent input change (so far)
            -- Get the transition dependent delay
            TmpDelay := VitalCalcDelay(NewValue, OldValue, Paths(i).PathDelay);

            -- If other inputs changed at the same time,
            -- then use the minimum of their propagation delays,
            -- else use the propagation delay from this input.
            IF Paths(i).InputChangeTime < InputAge THEN
                PropDelay := TmpDelay;
            ELSE -- Simultaneous inputs change
                IF TmpDelay < PropDelay THEN PropDelay := TmpDelay; END IF;
            end if;

            InputAge := Paths(i).InputChangeTime;
        END LOOP;

        -- If there were no paths (with an enabled condition),
        -- use the default delay, if so indicated, otherwise return TIME'HIGH
        IF (PropDelay = TIME'HIGH) THEN
  IF (IgnoreDefaultDelay) THEN
            PropDelay := VitalCalcDelay(NewValue, OldValue, DefaultDelay);
  END IF;

-- If the time since the most recent selected input event is
-- greater than the propagation delay from that input,
-- then use the default delay (won't happen if no paths are selected)
        ELSIF (InputAge > PropDelay) THEN
            PropDelay := VitalCalcDelay(NewValue, OldValue, DefaultDelay);

        -- Adjust the propagation delay by the time since the
        -- the input event occurred  (Usually 0 ns).
        ELSE
            PropDelay := PropDelay - InputAge;
        END IF;

        RETURN PropDelay;
    END;


    ---------------------------------------------------------------------------
    ---------------------------------------------------------------------------
    -- Glitch Handlers
    ---------------------------------------------------------------------------
    ---------------------------------------------------------------------------
    PROCEDURE ReportGlitch (
            CONSTANT GlitchRoutine  : IN  STRING;
            CONSTANT OutSignalName  : IN  STRING;
            CONSTANT PreemptedTime  : IN  TIME;
            CONSTANT PreemptedValue : IN  std_ulogic;
            CONSTANT NewTime        : IN  TIME;
            CONSTANT NewValue       : IN  std_ulogic;
            CONSTANT Index          : IN  INTEGER := 0;
            CONSTANT IsArraySignal  : IN  BOOLEAN := FALSE;
            CONSTANT MsgSeverity    : IN  SEVERITY_LEVEL := WARNING
    ) IS

        VARIABLE StrPtr1, StrPtr2, StrPtr3, StrPtr4, StrPtr5 : LINE;
    BEGIN

        Write (StrPtr1, PreemptedTime );
        Write (StrPtr2, NewTime);
        Write (StrPtr3, LogicCvtTable(PreemptedValue));
        Write (StrPtr4, LogicCvtTable(NewValue));
        IF IsArraySignal THEN
            Write (StrPtr5, STRING'( "(" ) );
            Write (StrPtr5, Index);
            Write (StrPtr5, STRING'( ")" ) );
        ELSE
            Write (StrPtr5, STRING'( " " ) );
        END IF;

        -- Issue Report only if Preempted value has not been
```

 Copyright © 2001 IEEE. All rights reserved.

```
        --  removed from event queue
        ASSERT PreemptedTime >  NewTime
          REPORT GlitchRoutine & ": GLITCH Detected on port " &
                 OutSignalName & StrPtr5.ALL &
                 "; Preempted Future Value := " & StrPtr3.ALL &
                 " @ " & StrPtr1.ALL &
                 "; Newly Scheduled Value := " & StrPtr4.ALL &
                 " @ " & StrPtr2.ALL &
                 ";"
          SEVERITY MsgSeverity;


  ASSERT PreemptedTime <= NewTime
          REPORT GlitchRoutine & ": GLITCH Detected on port " &
                 OutSignalName & StrPtr5.ALL &
                 "; Negative Preempted Value := " & StrPtr3.ALL &
                 " @ " & StrPtr1.ALL &
                 "; Newly Scheduled Value := " & StrPtr4.ALL &
                 " @ " & StrPtr2.ALL &
                 ";"
          SEVERITY MsgSeverity;


        DEALLOCATE(StrPtr1);
        DEALLOCATE(StrPtr2);
        DEALLOCATE(StrPtr3);
        DEALLOCATE(StrPtr4);
        DEALLOCATE(StrPtr5);
        RETURN;
    END ReportGlitch;

    ---------------------------------------------------------------------------
    PROCEDURE VitalGlitch (
        SIGNAL   OutSignal     : OUT   std_logic;
        VARIABLE GlitchData    : INOUT VitalGlitchDataType;
        CONSTANT OutSignalName : IN    string;
        CONSTANT NewValue      : IN    std_logic;
        CONSTANT NewDelay      : IN    TIME            := 0 ns;
        CONSTANT Mode          : IN    VitalGlitchKindType := OnEvent;
        CONSTANT XOn           : IN    BOOLEAN         := TRUE;
        CONSTANT NegPreemptOn  : IN    BOOLEAN         := FALSE; --IR225
        CONSTANT MsgOn         : IN    BOOLEAN         := FALSE;
        CONSTANT MsgSeverity   : IN    SEVERITY_LEVEL := WARNING
        ) IS
    ---------------------------------------------------------------------------
        VARIABLE NewGlitch : BOOLEAN := TRUE;
        VARIABLE dly       : TIME    := NewDelay;
        VARIABLE NOW_TIME  : TIME    := NOW;
        VARIABLE NegPreemptGlitch : BOOLEAN := FALSE;

    BEGIN
      NegPreemptGlitch:=FALSE;--reset Preempt-Glitch

        -- If nothing to schedule, just return
        IF NewDelay < 0 ns THEN
            IF (NewValue /= GlitchData.SchedValue) THEN
                VitalError ( "VitalGlitch", ErrNegDel, OutSignalName );
            END IF;
            RETURN;
        END IF;

        -- If simple signal assignment
        --   perform the signal assignment
        IF    ( Mode = VitalInertial) THEN
            OutSignal <= NewValue AFTER dly;
        ELSIF ( Mode = VitalTransport ) THEN
            OutSignal <= TRANSPORT NewValue AFTER dly;
        ELSE
          -- Glitch Processing ---
          -- If nothing currently scheduled
          IF GlitchData.SchedTime <= NOW THEN  -- NOW >= last event
             -- Note: NewValue is always /= OldValue when called from VPPD
            IF (NewValue = GlitchData.SchedValue) THEN RETURN; END IF;
      NewGlitch := FALSE;
          GlitchData.GlitchTime := NOW+dly;

          -- New value earlier than the earliest previous value scheduled
          --  (negative preemptive)
          ELSIF (NOW+dly <= GlitchData.GlitchTime)
             AND (NOW+dly <= GlitchData.SchedTime)  THEN
```

Copyright © 2001 IEEE. All rights reserved.

```
          -- Glitch is negative preemptive - check if same value and
          -- NegPreempt is on IR225
          IF (GlitchData.SchedValue /= NewValue) AND (NegPreemptOn) AND
             (NOW > 0 NS) THEN
            NewGlitch := TRUE;
            NegPreemptGlitch :=TRUE;-- Set preempt Glitch condition
          ELSE
            NewGlitch := FALSE;    -- No new glitch, save time for
                                   -- possible future glitch
          END IF;
          GlitchData.GlitchTime := NOW+dly;

          -- Transaction currently scheduled - if glitch already happened
          ELSIF GlitchData.GlitchTime <= NOW THEN
             IF (GlitchData.SchedValue = NewValue) THEN
                 dly := Minimum( GlitchData.SchedTime-NOW, NewDelay );
             END IF;
             NewGlitch := FALSE;

          -- Transaction currently scheduled (no glitch if same value)
          ELSIF (GlitchData.SchedValue = NewValue)
                AND (GlitchData.SchedTime = GlitchData.GlitchTime) THEN
              -- revise scheduled output time if new delay is sooner
              dly := Minimum( GlitchData.SchedTime-NOW, NewDelay );
              -- No new glitch, save time for possable future glitch
              NewGlitch := FALSE;
              GlitchData.GlitchTime := NOW+dly;

          -- Transaction currently scheduled represents a glitch
          ELSE
            NewGlitch := TRUE; -- A new glitch has been detected
          END IF;

          IF NewGlitch THEN
            -- If  messages requested, report the glitch
            IF MsgOn THEN
              IF NegPreemptGlitch THEN  --IR225
                ReportGlitch ("VitalGlitch-Neg", OutSignalName,
                               GlitchData.GlitchTime, GlitchData.SchedValue,
                               (dly + NOW), NewValue,
                               MsgSeverity=>MsgSeverity  );
              ELSE
            ReportGlitch ("VitalGlitch", OutSignalName,
                               GlitchData.GlitchTime, GlitchData.SchedValue,
                               (dly + NOW), NewValue,
                               MsgSeverity=>MsgSeverity  );
         END IF;
     END IF;

          -- If 'X' generation is requested, schedule the new value
          --  preceeded by a glitch pulse.
          -- Otherwise just schedule the new value (inertial mode).
          IF XOn THEN
            IF (Mode = OnDetect) THEN
              OutSignal <= 'X';
            ELSE
              OutSignal <= 'X' AFTER GlitchData.GlitchTime-NOW;
          END IF;

          IF NegPreemptGlitch THEN -- IR225
             OutSignal <= TRANSPORT NewValue AFTER GlitchData.SchedTime-NOW;
          ELSE
            OutSignal <= TRANSPORT NewValue AFTER dly;
          END IF;
          ELSE
            OutSignal <= NewValue AFTER dly; -- no glitch regular prop delay
          END IF;

          -- If there no new glitch was detected, just schedule the new value.
          ELSE
            OutSignal <= NewValue AFTER dly;
          END IF;
     END IF;

     -- Record the new value and time depending on glitch type just scheduled.
     IF NOT NegPreemptGlitch THEN -- 5/2/96 for "x-pulse" IR225
        GlitchData.SchedValue := NewValue;
        GlitchData.SchedTime  := NOW+dly;   -- pulse timing.
ELSE
      GlitchData.SchedValue := 'X';
      -- leave GlitchData.SchedTime to old value since glitch is negative
```

 Copyright © 2001 IEEE. All rights reserved.

```
      END IF;
    RETURN;
  END;

  ---------------------------------------------------------------------------
 PROCEDURE VitalPathDelay (
      SIGNAL   OutSignal          : OUT   std_logic;
      VARIABLE GlitchData         : INOUT VitalGlitchDataType;
      CONSTANT OutSignalName      : IN    string;
      CONSTANT OutTemp            : IN    std_logic;
      CONSTANT Paths              : IN    VitalPathArrayType;
      CONSTANT DefaultDelay       : IN    VitalDelayType      := VitalZeroDelay;
      CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
      CONSTANT XOn                : IN    BOOLEAN             := TRUE;
      CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
      CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
      CONSTANT NegPreemptOn       : IN    BOOLEAN             := FALSE;--IR2253/14/
98
    CONSTANT IgnoreDefaultDelay : IN   BOOLEAN          := FALSE      --IR248 3/14/98
  ) IS

      VARIABLE PropDelay : TIME;

  BEGIN
      -- Check if the new value to be scheduled is different than the
      -- previously scheduled value
      IF (GlitchData.SchedTime <= NOW) AND
         (GlitchData.SchedValue = OutTemp)
           THEN RETURN;
      END IF;

      -- Evaluate propagation delay paths
      PropDelay := VitalSelectPathDelay (OutTemp, GlitchData.LastValue,
                                         OutSignalName, Paths, DefaultDelay,
                     IgnoreDefaultDelay);

      GlitchData.LastValue := OutTemp;

      -- Schedule the output transactions - including glitch handling
      VitalGlitch (OutSignal, GlitchData, OutSignalName, OutTemp,
                   PropDelay, Mode, XOn, NegPreemptOn, MsgOn, MsgSeverity );

  END VitalPathDelay;

  ---------------------------------------------------------------------------

  PROCEDURE VitalPathDelay01 (
      SIGNAL   OutSignal          : OUT   std_logic;
      VARIABLE GlitchData         : INOUT VitalGlitchDataType;
      CONSTANT OutSignalName      : IN    string;
      CONSTANT OutTemp            : IN    std_logic;
      CONSTANT Paths              : IN    VitalPathArray01Type;
      CONSTANT DefaultDelay       : IN    VitalDelayType01    := VitalZeroDelay01;
      CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
      CONSTANT XOn                : IN    BOOLEAN             := TRUE;
      CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
      CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
      CONSTANT NegPreemptOn       : IN    BOOLEAN             := FALSE;--IR2253/14/
98
    CONSTANT IgnoreDefaultDelay : IN   BOOLEAN          := FALSE;     --IR248 3/14/98
      CONSTANT RejectFastPath     : IN    BOOLEAN             := FALSE        --IR250


  ) IS

      VARIABLE PropDelay : TIME;
  BEGIN

      -- Check if the new value to be scheduled is different than the
      -- previously scheduled value
      IF (GlitchData.SchedTime <= NOW) AND
         (GlitchData.SchedValue = OutTemp)
           THEN RETURN;
      -- Check if the new value to be Scheduled is the same as the
      -- previously scheduled output transactions.  If this condition
      -- exists and the new scheduled time is < the current GlitchData.
      -- schedTime then a fast path condition exists (IR250).  If the
      -- modeler wants this condition rejected by setting the
      -- RejectFastPath actual to true then exit out.
      ELSIF (GlitchData.SchedValue=OutTemp)  AND (RejectFastPath)
           THEN RETURN;
```

Copyright © 2001 IEEE. All rights reserved.

```
        END IF;

        -- Evaluate propagation delay paths
        PropDelay := VitalSelectPathDelay (OutTemp, GlitchData.LastValue,
                                           OutSignalName, Paths, DefaultDelay,
                     IgnoreDefaultDelay);

        GlitchData.LastValue := OutTemp;


        VitalGlitch (OutSignal, GlitchData, OutSignalName, OutTemp,
                     PropDelay, Mode, XOn, NegPreemptOn, MsgOn, MsgSeverity );
    END VitalPathDelay01;

    ---------------------------------------------------------------------------
    PROCEDURE VitalPathDelay01Z (
        SIGNAL   OutSignal           : OUT   std_logic;
        VARIABLE GlitchData          : INOUT VitalGlitchDataType;
        CONSTANT OutSignalName       : IN    string;
        CONSTANT OutTemp             : IN    std_logic;
        CONSTANT Paths               : IN    VitalPathArray01ZType;
        CONSTANT DefaultDelay        : IN    VitalDelayType01Z   := VitalZeroDelay01Z;
        CONSTANT Mode                : IN    VitalGlitchKindType := OnEvent;
        CONSTANT XOn                 : IN    BOOLEAN             := TRUE;
        CONSTANT MsgOn               : IN    BOOLEAN             := TRUE;
        CONSTANT MsgSeverity         : IN    SEVERITY_LEVEL      := WARNING;
        CONSTANT OutputMap           : IN    VitalOutputMapType:=
VitalDefaultOutputMap;
        CONSTANT NegPreemptOn        : IN    BOOLEAN             := FALSE;--IR2253/14/
98
      CONSTANT IgnoreDefaultDelay : IN    BOOLEAN           := FALSE;      --IR248 3/14/98
        CONSTANT RejectFastPath      : IN    BOOLEAN             := FALSE        --IR250
     ) IS

        VARIABLE PropDelay : TIME;

    BEGIN
       -- Check if the new value to be scheduled is different than the
       -- previously scheduled value
        IF (GlitchData.SchedTime <= NOW) AND
           (GlitchData.SchedValue = OutTemp)
             THEN RETURN;
       -- Check if the new value to be Scheduled is the same as the
       -- previously scheduled output transactions.  If this condition
       -- exists and the new scheduled time is < the current GlitchData.
       -- schedTime then a fast path condition exists (IR250).  If the
       -- modeler wants this condition rejected by setting the
       -- RejectFastPath actual to true then exit out.
        ELSIF (GlitchData.SchedValue=OutTemp)  AND (RejectFastPath)
             THEN RETURN;
        END IF;

        -- Evaluate propagation delay paths
        PropDelay := VitalSelectPathDelay (OutTemp, GlitchData.LastValue,
                                           OutSignalName, Paths, DefaultDelay,
                     IgnoreDefaultDelay);

        GlitchData.LastValue := OutTemp;


        -- Schedule the output transactions - including glitch handling
          VitalGlitch (OutSignal, GlitchData, OutSignalName, OutTemp,
                     PropDelay, Mode, XOn, NegPreemptOn, MsgOn, MsgSeverity );
    END VitalPathDelay01Z;


    ----------------------------------------------------------------------------
    PROCEDURE VitalWireDelay (
            SIGNAL   OutSig  : OUT   std_ulogic;
            SIGNAL   InSig   : IN    std_ulogic;
            CONSTANT twire   : IN    VitalDelayType
    ) IS
    BEGIN
        OutSig <= TRANSPORT InSig AFTER twire;
    END VitalWireDelay;

    PROCEDURE VitalWireDelay (
            SIGNAL   OutSig  : OUT   std_ulogic;
            SIGNAL   InSig   : IN    std_ulogic;
            CONSTANT twire   : IN    VitalDelayType01
    ) IS
```

 Copyright © 2001 IEEE. All rights reserved.

```
   VARIABLE Delay   : TIME;
BEGIN
    Delay := VitalCalcDelay( InSig, InSig'LAST_VALUE, twire );
    OutSig <= TRANSPORT InSig AFTER Delay;
END VitalWireDelay;

PROCEDURE VitalWireDelay (
        SIGNAL   OutSig  : OUT   std_ulogic;
        SIGNAL   InSig   : IN    std_ulogic;
        CONSTANT twire   : IN    VitalDelayType01Z
) IS
   VARIABLE Delay   : TIME;
BEGIN
    Delay := VitalCalcDelay( InSig, InSig'LAST_VALUE, twire );
    OutSig <= TRANSPORT InSig AFTER Delay;
END VitalWireDelay;

----------------------------------------------------------------------------
PROCEDURE VitalSignalDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT dly        : IN   TIME
) IS
BEGIN
    OutSig <= TRANSPORT InSig AFTER dly;
END;

---------------------------------------------------------------------------
---------------------------------------------------------------------------
-- Setup and Hold Time Check Routine
---------------------------------------------------------------------------
---------------------------------------------------------------------------
PROCEDURE ReportViolation (
        CONSTANT TestSignalName : IN STRING := "";
        CONSTANT RefSignalName  : IN STRING := "";
        CONSTANT HeaderMsg      : IN STRING := " ";
        CONSTANT CheckInfo      : IN CheckInfoType;
        CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
) IS
    VARIABLE Message : LINE;
BEGIN
    IF NOT CheckInfo.Violation THEN RETURN; END IF;

    Write ( Message, HeaderMsg );
    Case CheckInfo.CheckKind IS
      WHEN    SetupCheck => Write ( Message, STRING'(" SETUP ")      );
      WHEN     HoldCheck => Write ( Message, STRING'(" HOLD ")       );
      WHEN RecoveryCheck => Write ( Message, STRING'(" RECOVERY ")   );
      WHEN  RemovalCheck => Write ( Message, STRING'(" REMOVAL ")    );
      WHEN PulseWidCheck => Write ( Message, STRING'(" PULSE WIDTH "));
      WHEN   PeriodCheck => Write ( Message, STRING'(" PERIOD ")     );
    END CASE;
    Write ( Message, HiLoStr(CheckInfo.State) );
    Write ( Message, STRING'(" VIOLATION ON ") );
    Write ( Message, TestSignalName );
    IF (RefSignalName'LENGTH > 0) THEN
        Write ( Message, STRING'(" WITH RESPECT TO ") );
        Write ( Message, RefSignalName );
    END IF;
    Write ( Message, ';' & LF );
    Write ( Message, STRING'("  Expected := ")  );
    Write ( Message, CheckInfo.ExpTime);
    Write ( Message, STRING'("; Observed := ")  );
    Write ( Message, CheckInfo.ObsTime);
    Write ( Message, STRING'("; At : ")         );
    Write ( Message, CheckInfo.DetTime);

    ASSERT FALSE REPORT Message.ALL SEVERITY MsgSeverity;

    DEALLOCATE (Message);
END ReportViolation;


---------------------------------------------------------------------------
-- Procedure  : InternalTimingCheck
---------------------------------------------------------------------------
PROCEDURE InternalTimingCheck (
        CONSTANT TestSignal    : IN     std_ulogic;
        CONSTANT RefSignal     : IN     std_ulogic;
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        CONSTANT RefDelay      : IN     TIME := 0 ns;
```

```
            CONSTANT SetupHigh     : IN     TIME := 0 ns;
            CONSTANT SetupLow      : IN     TIME := 0 ns;
            CONSTANT HoldHigh      : IN     TIME := 0 ns;
            CONSTANT HoldLow       : IN     TIME := 0 ns;
            VARIABLE RefTime       : IN     TIME;
            VARIABLE RefEdge       : IN     BOOLEAN;
            VARIABLE TestTime      : IN     TIME;
            VARIABLE TestEvent     : IN     BOOLEAN;
            VARIABLE SetupEn       : INOUT  BOOLEAN;
            VARIABLE HoldEn        : INOUT  BOOLEAN;
            VARIABLE CheckInfo     : INOUT  CheckInfoType;
            CONSTANT MsgOn         : IN     BOOLEAN
    ) IS
        VARIABLE bias : TIME;
VARIABLE actualObsTime : TIME;
        VARIABLE BC : TIME;
VARIABLE Message:LINE;
    BEGIN
        -- Check SETUP constraint
      IF RefEdge THEN
          IF SetupEn THEN
            CheckInfo.ObsTime   := RefTime - TestTime;
            CheckInfo.State     := To_X01(TestSignal);
            CASE CheckInfo.State IS
              WHEN '0'  => CheckInfo.ExpTime := SetupLow;
         -- start of new code IR245-246
           BC := HoldHigh;
         -- end of new code IR245-246
              WHEN '1'  => CheckInfo.ExpTime := SetupHigh;
         -- start of new code IR245-246
           BC := HoldLow;
         -- end of new code IR245-246
              WHEN 'X'  => CheckInfo.ExpTime := Maximum(SetupHigh,SetupLow);
         -- start of new code IR245-246
           BC := Maximum(HoldHigh,HoldLow);
         -- end of new code IR245-246
            END CASE;
       -- added the second condition for IR 245-246
       CheckInfo.Violation := ( (CheckInfo.ObsTime < CheckInfo.ExpTime)
           AND ( NOT ((CheckInfo.ObsTime = BC) and (BC = 0 ns))) );
       -- start of new code IR245-246
       IF(CheckInfo.ExpTime = 0 ns) THEN
           CheckInfo.CheckKind := HoldCheck;
       ELSE
           CheckInfo.CheckKind := SetupCheck;
       END IF;
       -- end of new code IR245-246
       SetupEn  := FALSE;
          ELSE
            CheckInfo.Violation := FALSE;
          END IF;

        -- Check HOLD constraint
      ELSIF TestEvent THEN
          IF HoldEn THEN
            CheckInfo.ObsTime := TestTime - RefTime;
            CheckInfo.State := To_X01(TestSignal);
            CASE CheckInfo.State IS
              WHEN '0'  => CheckInfo.ExpTime := HoldHigh;

         -- new code for unnamed IR
            CheckInfo.State := '1';

         -- start of new code IR245-246
           BC := SetupLow;
         -- end of new code IR245-246
              WHEN '1'  => CheckInfo.ExpTime := HoldLow;

         -- new code for unnamed IR
            CheckInfo.State := '0';

         -- start of new code IR245-246
           BC := SetupHigh;
         -- end of new code IR245-246
              WHEN 'X'  => CheckInfo.ExpTime := Maximum(HoldHigh,HoldLow);
         -- start of new code IR245-246
           BC := Maximum(SetupHigh,SetupLow);
         -- end of new code IR245-246
            END CASE;
       -- added the second condition for IR 245-246
       CheckInfo.Violation := ( (CheckInfo.ObsTime < CheckInfo.ExpTime)
```


Copyright © 2001 IEEE. All rights reserved.

```
            AND ( NOT ((CheckInfo.ObsTime = BC) and (BC = 0 ns))) );

        -- start of new code IR245-246
        IF(CheckInfo.ExpTime = 0 ns) THEN
            CheckInfo.CheckKind := SetupCheck;
        ELSE
            CheckInfo.CheckKind := HoldCheck;
        END IF;
        -- end of new code IR245-246
            HoldEn := NOT CheckInfo.Violation;
          ELSE
            CheckInfo.Violation := FALSE;
          END IF;
      ELSE
          CheckInfo.Violation := FALSE;
      END IF;

      -- Adjust report values to account for internal model delays
      -- Note: TestDelay, RefDelay, TestTime, RefTime are non-negative
      -- Note: bias may be negative or positive
      IF MsgOn AND CheckInfo.Violation THEN
      -- modified the code for correct reporting of violation in case of
      -- order of signals being reversed because of internal delays
-- new variable
actualObsTime := (TestTime-TestDelay)-(RefTime-RefDelay);
bias := TestDelay - RefDelay;
IF (actualObsTime < 0 ns) THEN -- It should be a setup check
          IF ( CheckInfo.CheckKind = HoldCheck) then
              CheckInfo.CheckKind := SetupCheck;
              CASE CheckInfo.State IS
                WHEN '0' => CheckInfo.ExpTime := SetupLow;
                WHEN '1' => CheckInfo.ExpTime := SetupHigh;
                WHEN 'X' => CheckInfo.ExpTime := Maximum(SetupHigh,SetupLow);
              END CASE;
          END IF;

  CheckInfo.ObsTime := -actualObsTime;
  CheckInfo.ExpTime := CheckInfo.ExpTime + bias;
  CheckInfo.DetTime := RefTime - RefDelay;
        ELSE -- It should be a hold check
         IF ( CheckInfo.CheckKind = SetupCheck) then
   CheckInfo.CheckKind :=  HoldCheck;
   CASE CheckInfo.State IS
     WHEN '0' => CheckInfo.ExpTime := HoldHigh;
        CheckInfo.State   := '1';
     WHEN '1' => CheckInfo.ExpTime := HoldLow;
        CheckInfo.State   := '0';
     WHEN 'X' => CheckInfo.ExpTime := Maximum(HoldHigh,HoldLow);
   END CASE;
         END IF;

 CheckInfo.ObsTime := actualObsTime;
 CheckInfo.ExpTime := CheckInfo.ExpTime - bias;
 CheckInfo.DetTime := TestTime - TestDelay;
END IF;

      END IF;
    END InternalTimingCheck;

    ---------------------------------------------------------------------------
    ---------------------------------------------------------------------------
    FUNCTION VitalTimingDataInit
            RETURN VitalTimingDataType IS
    BEGIN
        RETURN (FALSE,'X', 0 ns, FALSE, 'X', 0 ns, FALSE, NULL, NULL, NULL, NULL);
    END;

    ---------------------------------------------------------------------------
    -- Procedure  : VitalSetupHoldCheck
    ---------------------------------------------------------------------------
    PROCEDURE VitalSetupHoldCheck (
            VARIABLE Violation     : OUT    X01;
            VARIABLE TimingData    : INOUT  VitalTimingDataType;
            SIGNAL   TestSignal    : IN     std_ulogic;
            CONSTANT TestSignalName: IN     STRING := "";
            CONSTANT TestDelay     : IN     TIME := 0 ns;
            SIGNAL   RefSignal     : IN     std_ulogic;
            CONSTANT RefSignalName : IN     STRING := "";
            CONSTANT RefDelay      : IN     TIME := 0 ns;
            CONSTANT SetupHigh     : IN     TIME := 0 ns;
            CONSTANT SetupLow      : IN     TIME := 0 ns;
```

Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT HoldHigh      : IN     TIME := 0 ns;
        CONSTANT HoldLow       : IN     TIME := 0 ns;
        CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg     : IN     STRING  := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;  --IR252 3/23/98
    CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE--IR252 3/23/98
) IS

    VARIABLE CheckInfo : CheckInfoType;
    VARIABLE RefEdge, TestEvent : BOOLEAN;
    VARIABLE TestDly : TIME := Maximum(0 ns, TestDelay);
    VARIABLE RefDly  : TIME := Maximum(0 ns, RefDelay);
    VARIABLE bias : TIME;
BEGIN

  IF (TimingData.NotFirstFlag = FALSE) THEN
     TimingData.TestLast := To_X01(TestSignal);
     TimingData.RefLast := To_X01(RefSignal);
     TimingData.NotFirstFlag := TRUE;
  END IF;

  -- Detect reference edges and record the time of the last edge
  RefEdge := EdgeSymbolMatch(TimingData.RefLast, To_X01(RefSignal),
                             RefTransition);
  TimingData.RefLast := To_X01(RefSignal);
  IF RefEdge THEN
    TimingData.RefTime := NOW;
TimingData.SetupEn := TimingData.SetupEn AND EnableSetupOnRef;  --IR252 3/23/98
    TimingData.HoldEn  := EnableHoldOnRef;    --IR252 3/23/98
  END IF;

  -- Detect test (data) changes and record the time of the last change
  TestEvent := TimingData.TestLast /= To_X01Z(TestSignal);
  TimingData.TestLast := To_X01Z(TestSignal);
  IF TestEvent THEN
    TimingData.TestTime := NOW;
    TimingData.SetupEn  := EnableSetupOnTest;--IR252 3/23/98
TimingData.HoldEn   := TimingData.HoldEn AND EnableHoldOnTest;--IR252 3/23/98
  END IF;

  -- Perform timing checks (if enabled)
  Violation := '0';
  IF (CheckEnabled) THEN
     InternalTimingCheck (
         TestSignal    => TestSignal,
         RefSignal     => RefSignal,
         TestDelay     => TestDly,
         RefDelay      => RefDly,
         SetupHigh     => SetupHigh,
         SetupLow      => SetupLow,
         HoldHigh      => HoldHigh,
         HoldLow       => HoldLow,
         RefTime       => TimingData.RefTime,
         RefEdge       => RefEdge,
         TestTime      => TimingData.TestTime,
         TestEvent     => TestEvent,
         SetupEn       => TimingData.SetupEn,
         HoldEn        => TimingData.HoldEn,
         CheckInfo     => CheckInfo,
         MsgOn         => MsgOn );

    -- Report any detected violations and set return violation flag
    IF CheckInfo.Violation THEN
        IF (MsgOn) THEN
            ReportViolation (TestSignalName, RefSignalName,
                             HeaderMsg, CheckInfo, MsgSeverity );
        END IF;
        IF (XOn) THEN Violation := 'X'; END IF;
    END IF;
  END IF;

END VitalSetupHoldCheck;

---------------------------------------------------------------------------
PROCEDURE VitalSetupHoldCheck (
```

 Copyright © 2001 IEEE. All rights reserved.

```
        VARIABLE Violation     : OUT    X01;
        VARIABLE TimingData    : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal    : IN     std_logic_vector;
        CONSTANT TestSignalName: IN     STRING := "";
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        SIGNAL   RefSignal     : IN     std_ulogic;
        CONSTANT RefSignalName : IN     STRING := "";
        CONSTANT RefDelay      : IN     TIME := 0 ns;
        CONSTANT SetupHigh     : IN     TIME := 0 ns;
        CONSTANT SetupLow      : IN     TIME := 0 ns;
        CONSTANT HoldHigh      : IN     TIME := 0 ns;
        CONSTANT HoldLow       : IN     TIME := 0 ns;
        CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg     : IN     STRING := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;  --IR252 3/23/98
    CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE--IR252 3/23/98

) IS

    VARIABLE CheckInfo : CheckInfoType;
    VARIABLE RefEdge : BOOLEAN;
    VARIABLE TestEvent : VitalBoolArrayT(TestSignal'RANGE);
    VARIABLE TestDly : TIME := Maximum(0 ns, TestDelay);
    VARIABLE RefDly  : TIME := Maximum(0 ns, RefDelay);
    VARIABLE bias : TIME;
    VARIABLE ChangedAllAtOnce : BOOLEAN := TRUE;
    VARIABLE StrPtr1 : LINE;

BEGIN
  -- Initialization of working area.
  IF (TimingData.NotFirstFlag = FALSE) THEN
    TimingData.TestLastA := NEW std_logic_vector(TestSignal'RANGE);
    TimingData.TestTimeA := NEW  VitalTimeArrayT(TestSignal'RANGE);
    TimingData.HoldEnA   := NEW  VitalBoolArrayT(TestSignal'RANGE);
    TimingData.SetupEnA  := NEW  VitalBoolArrayT(TestSignal'RANGE);
    FOR i IN TestSignal'RANGE LOOP
      TimingData.TestLastA(i) := To_X01(TestSignal(i));
    END LOOP;
    TimingData.RefLast := To_X01(RefSignal);
    TimingData.NotFirstFlag := TRUE;
  END IF;

  -- Detect reference edges and record the time of the last edge
  RefEdge := EdgeSymbolMatch(TimingData.RefLast, To_X01(RefSignal),
                             RefTransition);
  TimingData.RefLast := To_X01(RefSignal);
  IF RefEdge THEN
    TimingData.RefTime := NOW;
TimingData.SetupEn := TimingData.SetupEn AND EnableSetupOnRef;--IR252 3/23/98
    TimingData.HoldEnA.all := (TestSignal'RANGE => EnableHoldOnRef);--IR252  3/23/
98
  END IF;

  -- Detect test (data) changes and record the time of the last change
  FOR i IN TestSignal'RANGE LOOP
    TestEvent(i) := TimingData.TestLastA(i) /= To_X01Z(TestSignal(i));
    TimingData.TestLastA(i) := To_X01Z(TestSignal(i));
    IF TestEvent(i) THEN
      TimingData.TestTimeA(i) := NOW;
      TimingData.SetupEnA(i) := EnableSetupOnTest;--IR252 3/23/98
    TimingData.HoldEnA(i) := TimingData.HoldEn AND EnableHoldOnTest;--IR252  3/23/
98
      TimingData.TestTime := NOW;                              --IR252 3/23/98
    END IF;
  END LOOP;

  -- Check to see if the Bus subelements changed all at the same time.
  -- If so, then we can reduce the volume of error messages since we no
  -- longer have to report every subelement individually
  FOR i IN TestSignal'RANGE LOOP
    IF TimingData.TestTimeA(i) /= TimingData.TestTime THEN
        ChangedAllAtOnce := FALSE;
        EXIT;
    END IF;
  END LOOP;
```

Copyright © 2001 IEEE. All rights reserved.

```
  -- Perform timing checks (if enabled)
  Violation := '0';
  IF (CheckEnabled) THEN
    FOR i IN TestSignal'RANGE LOOP
      InternalTimingCheck (
          TestSignal   => TestSignal(i),
          RefSignal    => RefSignal,
          TestDelay    => TestDly,
          RefDelay     => RefDly,
          SetupHigh    => SetupHigh,
          SetupLow     => SetupLow,
          HoldHigh     => HoldHigh,
          HoldLow      => HoldLow,
          RefTime      => TimingData.RefTime,
          RefEdge      => RefEdge,
          TestTime     => TimingData.TestTimeA(i),
          TestEvent    => TestEvent(i),
          SetupEn      => TimingData.SetupEnA(i),
          HoldEn       => TimingData.HoldEnA(i),
          CheckInfo    => CheckInfo,
          MsgOn        => MsgOn );

      -- Report any detected violations and set return violation flag
      IF CheckInfo.Violation THEN
        IF (MsgOn) THEN
          IF ( ChangedAllAtOnce AND (i = TestSignal'LEFT) ) THEN
            ReportViolation (TestSignalName&"(...)", RefSignalName,
                             HeaderMsg, CheckInfo, MsgSeverity );
          ELSIF (NOT ChangedAllAtOnce) THEN
            Write (StrPtr1, i);
            ReportViolation (TestSignalName & "(" & StrPtr1.ALL & ")",
                             RefSignalName,
                             HeaderMsg, CheckInfo, MsgSeverity );
            DEALLOCATE (StrPtr1);
          END IF;
        END IF;
        IF (XOn) THEN
          Violation := 'X';
        END IF;
      END IF;
    END LOOP;
  END IF;

  DEALLOCATE (StrPtr1);

END VitalSetupHoldCheck;

---------------------------------------------------------------------------
-- Function   : VitalRecoveryRemovalCheck
---------------------------------------------------------------------------
PROCEDURE VitalRecoveryRemovalCheck (
        VARIABLE Violation     : OUT    X01;
        VARIABLE TimingData    : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal    : IN     std_ulogic;
        CONSTANT TestSignalName: IN     STRING := "";
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        SIGNAL   RefSignal     : IN     std_ulogic;
        CONSTANT RefSignalName : IN     STRING := "";
        CONSTANT RefDelay      : IN     TIME := 0 ns;
        CONSTANT Recovery      : IN     TIME := 0 ns;
        CONSTANT Removal       : IN     TIME := 0 ns;
        CONSTANT ActiveLow     : IN     BOOLEAN := TRUE;
        CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg     : IN     STRING := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT EnableRecOnTest : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableRecOnRef  : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableRemOnRef  : IN   BOOLEAN := TRUE;--IR252 3/23/98
    CONSTANT EnableRemOnTest : IN   BOOLEAN := TRUE--IR252 3/23/98
    ) IS
    VARIABLE CheckInfo : CheckInfoType;
    VARIABLE RefEdge, TestEvent : BOOLEAN;
    VARIABLE TestDly : TIME := Maximum(0 ns, TestDelay);
    VARIABLE RefDly  : TIME := Maximum(0 ns, RefDelay);
    VARIABLE bias : TIME;
BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
    IF (TimingData.NotFirstFlag = FALSE) THEN
       TimingData.TestLast := To_X01(TestSignal);
       TimingData.RefLast := To_X01(RefSignal);
       TimingData.NotFirstFlag := TRUE;
    END IF;

    -- Detect reference edges and record the time of the last edge
    RefEdge := EdgeSymbolMatch(TimingData.RefLast, To_X01(RefSignal),
                               RefTransition);
    TimingData.RefLast := To_X01(RefSignal);
    IF RefEdge THEN
      TimingData.RefTime := NOW;
  TimingData.SetupEn := TimingData.SetupEn AND EnableRecOnRef;  --IR252 3/23/98
      TimingData.HoldEn  := EnableRemOnRef;  --IR252 3/23/98
    END IF;

    -- Detect test (data) changes and record the time of the last change
    TestEvent := TimingData.TestLast /= To_X01Z(TestSignal);
    TimingData.TestLast := To_X01Z(TestSignal);
    IF TestEvent THEN
      TimingData.TestTime := NOW;
      TimingData.SetupEn  := EnableRecOnTest;  --IR252 3/23/98
  TimingData.HoldEn   := TimingData.HoldEn AND EnableRemOnTest;  --IR252 3/23/98
    END IF;

    -- Perform timing checks (if enabled)
    Violation := '0';
    IF (CheckEnabled) THEN

      IF ActiveLow THEN
       InternalTimingCheck (
           TestSignal, RefSignal, TestDly, RefDly,
           Recovery, 0 ns, 0 ns, Removal,
           TimingData.RefTime, RefEdge,
           TimingData.TestTime, TestEvent,
           TimingData.SetupEn, TimingData.HoldEn,
           CheckInfo, MsgOn );
      ELSE
       InternalTimingCheck (
           TestSignal, RefSignal, TestDly, RefDly,
           0 ns, Recovery, Removal, 0 ns,
           TimingData.RefTime, RefEdge,
           TimingData.TestTime, TestEvent,
           TimingData.SetupEn, TimingData.HoldEn,
           CheckInfo, MsgOn );
      END IF;


      -- Report any detected violations and set return violation flag
      IF CheckInfo.Violation THEN
          IF CheckInfo.CheckKind = SetupCheck THEN
              CheckInfo.CheckKind := RecoveryCheck;
          ELSE
              CheckInfo.CheckKind := RemovalCheck;
          END IF;
          IF (MsgOn) THEN
              ReportViolation (TestSignalName, RefSignalName,
                               HeaderMsg, CheckInfo, MsgSeverity );
          END IF;
          IF (XOn) THEN Violation := 'X'; END IF;
      END IF;
    END IF;

  END VitalRecoveryRemovalCheck;

  ---------------------------------------------------------------------------
  PROCEDURE VitalPeriodPulseCheck  (
          VARIABLE Violation      : OUT    X01;
          VARIABLE PeriodData     : INOUT  VitalPeriodDataType;
          SIGNAL   TestSignal     : IN     std_ulogic;
          CONSTANT TestSignalName : IN     STRING := "";
          CONSTANT TestDelay      : IN     TIME := 0 ns;
          CONSTANT Period         : IN     TIME := 0 ns;
          CONSTANT PulseWidthHigh : IN     TIME := 0 ns;
          CONSTANT PulseWidthLow  : IN     TIME := 0 ns;
          CONSTANT CheckEnabled   : IN     BOOLEAN := TRUE;
          CONSTANT HeaderMsg      : IN     STRING := " ";
          CONSTANT XOn            : IN     BOOLEAN := TRUE;
          CONSTANT MsgOn          : IN     BOOLEAN := TRUE;
          CONSTANT MsgSeverity    : IN     SEVERITY_LEVEL := WARNING
  ) IS
```

Copyright © 2001 IEEE. All rights reserved.

```
        VARIABLE TestDly : TIME := Maximum(0 ns, TestDelay);
        VARIABLE CheckInfo : CheckInfoType;
        VARIABLE PeriodObs : TIME;
        VARIABLE PulseTest, PeriodTest : BOOLEAN;
        VARIABLE TestValue : X01 := To_X01(TestSignal);
    BEGIN

        IF (PeriodData.NotFirstFlag = FALSE) THEN
           PeriodData.Rise :=
                -maximum(Period, maximum(PulseWidthHigh, PulseWidthLow));
           PeriodData.Fall :=
                -maximum(Period, maximum(PulseWidthHigh, PulseWidthLow));
           PeriodData.Last := To_X01(TestSignal);
           PeriodData.NotFirstFlag := TRUE;
        END IF;

        -- Initialize for no violation
        -- No violation possible if no test signal change
        Violation := '0';
        IF (PeriodData.Last = TestValue) THEN
            RETURN;
        END IF;

        -- record starting pulse times
        IF EdgeSymbolMatch(PeriodData.Last, TestValue, 'P') THEN
            -- Compute period times, then record the High Rise Time
            PeriodObs := NOW - PeriodData.Rise;
            PeriodData.Rise := NOW;
            PeriodTest := TRUE;
        ELSIF EdgeSymbolMatch(PeriodData.Last, TestValue, 'N') THEN
            -- Compute period times, then record the Low Fall Time
            PeriodObs := NOW - PeriodData.Fall;
            PeriodData.Fall := NOW;
            PeriodTest := TRUE;
        ELSE
            PeriodTest := FALSE;
END IF;

        -- do checks on pulse ends
        IF EdgeSymbolMatch(PeriodData.Last, TestValue, 'p') THEN
            -- Compute pulse times
            CheckInfo.ObsTime := NOW - PeriodData.Fall;
            CheckInfo.ExpTime := PulseWidthLow;
            PulseTest := TRUE;
        ELSIF EdgeSymbolMatch(PeriodData.Last, TestValue, 'n') THEN
            -- Compute pulse times
            CheckInfo.ObsTime := NOW - PeriodData.Rise;
            CheckInfo.ExpTime := PulseWidthHigh;
            PulseTest := TRUE;
        ELSE
            PulseTest := FALSE;
        END IF;

        IF PulseTest AND CheckEnabled THEN
           -- Verify Pulse Width [ignore 1st edge]
            IF ( CheckInfo.ObsTime < CheckInfo.ExpTime ) THEN
                IF (XOn) THEN Violation := 'X'; END IF;
                IF (MsgOn) THEN
                    CheckInfo.Violation := TRUE;
                    CheckInfo.CheckKind := PulseWidCheck;
                    CheckInfo.DetTime   := NOW - TestDly;
                    CheckInfo.State     := PeriodData.Last;
                    ReportViolation (TestSignalName, "",
                                     HeaderMsg, CheckInfo, MsgSeverity );
                END IF; -- MsgOn
            END IF;
        END IF;

        IF PeriodTest AND CheckEnabled THEN
            -- Verify the Period [ignore 1st edge]
            CheckInfo.ObsTime := PeriodObs;
            CheckInfo.ExpTime := Period;
            IF ( CheckInfo.ObsTime < CheckInfo.ExpTime ) THEN
                IF (XOn) THEN Violation := 'X'; END IF;
                IF (MsgOn) THEN
                    CheckInfo.Violation := TRUE;
                    CheckInfo.CheckKind := PeriodCheck;
                    CheckInfo.DetTime   := NOW - TestDly;
                    CheckInfo.State     := TestValue;
                    ReportViolation (TestSignalName, "",
```

 Copyright © 2001 IEEE. All rights reserved.

```
                                HeaderMsg, CheckInfo, MsgSeverity );
            END IF; -- MsgOn
        END IF;
    END IF;

    PeriodData.Last := TestValue;

END VitalPeriodPulseCheck;



PROCEDURE ReportSkewViolation (
    CONSTANT Signal1Name    : IN STRING := "";
    CONSTANT Signal2Name    : IN STRING := "";
    CONSTANT ExpectedTime   : IN TIME;
    CONSTANT OccuranceTime  : IN TIME;
    CONSTANT HeaderMsg      : IN STRING;
    CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING;
    CONSTANT SkewPhase      : IN SkewType;
    CONSTANT ViolationFlag : IN BOOLEAN := TRUE
) IS
    VARIABLE Message : LINE;
BEGIN
    Write ( Message, HeaderMsg );
    IF (ViolationFlag /= TRUE) THEN
      Write ( Message, STRING'(" POSSIBLE") );
    END IF;
    IF (SkewPhase = Inphase) THEN
      Write ( Message, STRING'(" IN PHASE ")      );
    ELSE
      Write ( Message, STRING'(" OUT OF PHASE ")      );
    END IF;
    Write ( Message, STRING'("SKEW VIOLATION ON ")      );
    Write ( Message, Signal2Name );
    IF (Signal1Name'LENGTH > 0) THEN
        Write ( Message, STRING'(" WITH RESPECT TO ") );
        Write ( Message, Signal1Name );
    END IF;
    Write ( Message, ';' & LF );
    Write ( Message, STRING'(" At : ")         );
    Write ( Message, OccuranceTime);
    Write ( Message, STRING'("; Skew Limit : ")  );
    Write ( Message, ExpectedTime);

    ASSERT FALSE REPORT Message.ALL SEVERITY MsgSeverity;

    DEALLOCATE (Message);
END ReportSkewViolation;


PROCEDURE VitalInPhaseSkewCheck (
    VARIABLE Violation         : OUT    X01;
    VARIABLE SkewData          : INOUT  VitalSkewDataType;
    SIGNAL   Signal1           : IN     std_ulogic;
    CONSTANT Signal1Name       : IN     STRING := "";
    CONSTANT Signal1Delay      : IN     TIME := 0 ns;
    SIGNAL   Signal2           : IN     std_ulogic;
    CONSTANT Signal2Name       : IN     STRING := "";
    CONSTANT Signal2Delay      : IN     TIME := 0 ns;
    CONSTANT SkewS1S2RiseRise  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS2S1RiseRise  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS1S2FallFall  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS2S1FallFall  : IN     TIME := TIME'HIGH;
    CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
    CONSTANT XOn               : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT HeaderMsg         : IN     STRING  := "";
    SIGNAL   Trigger           : INOUT  std_ulogic
) IS
    VARIABLE ReportType  : VitalSkewExpectedType := none;
    VARIABLE ExpectedType  : VitalSkewExpectedType := none;
    VARIABLE ReportTime  : TIME;
    VARIABLE TriggerDelay  : TIME;
    VARIABLE ViolationCertain  : Boolean := TRUE;
BEGIN
    Violation := '0';
    ReportType := none;
    TriggerDelay := noTrigger;

    IF (CheckEnabled) THEN
```


Copyright © 2001 IEEE. All rights reserved.

```
IF (SkewData.ExpectedType /= none) THEN
    IF (trigger'Event) THEN
        CASE SkewData.ExpectedType IS
        WHEN s1r => ReportType := s1r;
                    ReportTime := NOW - Signal1Delay;
        WHEN s1f => ReportType := s1f;
                    ReportTime := NOW - Signal1Delay;
        WHEN s2r => ReportType := s2r;
                    ReportTime := NOW - Signal2Delay;
        WHEN s2f => ReportType := s2f;
                    ReportTime := NOW - Signal2Delay;
        WHEN OTHERS =>
        END CASE;
        SkewData.ExpectedType := none;
    ELSIF ( Signal1'Event OR Signal2'Event ) THEN
        IF ( Signal1 /= 'X' AND Signal2 /= 'X' ) THEN
            TriggerDelay := 0 ns;
            ExpectedType := none;
        END IF;
    END IF;
END IF;

IF (Signal1'EVENT and Signal2'EVENT) THEN
    IF (Signal1 = Signal2) THEN
        IF (Posedge(Signal1'LAST_VALUE, Signal1)) THEN
            IF ((Signal1Delay - Signal2Delay) >=
                    SkewS1S2RiseRise) THEN
                ReportType := s2r;
                ReportTime := NOW - Signal1Delay +
                              SkewS1S2RiseRise;
            ELSIF ((Signal2Delay -Signal1Delay) >=
                    SkewS2S1RiseRise) THEN
                ReportType := s1r;
                ReportTime := NOW - Signal2Delay +
                              SkewS2S1RiseRise;
            END IF;
        ELSIF (Negedge(Signal1'LAST_VALUE, Signal1)) THEN
            IF ((Signal1Delay - Signal2Delay) >=
                    SkewS1S2FallFall) THEN
                ReportType := s2f;
                ReportTime := NOW - Signal1Delay +
                              SkewS1S2FallFall;
            ELSIF ((Signal2Delay - Signal1Delay) >=
                    SkewS2S1FallFall) THEN
                ReportType := s1f;
                ReportTime := NOW - Signal2Delay +
                              SkewS2S1FallFall;
            END IF;
        END IF;
    ELSIF (Posedge(Signal1'LAST_VALUE , Signal1)) THEN
        IF ((Signal1Delay >= Signal2Delay) and (Signal2Delay >
                SkewS2S1FallFall)) THEN
            ReportType := s1f;
            ReportTime := NOW - Signal2Delay +
                                   SkewS2S1FallFall;
        ELSIF ((Signal2Delay >= Signal1Delay) and (Signal1Delay >
                SkewS1S2RiseRise)) THEN
            ReportType := s2r;
            ReportTime := NOW - Signal1Delay +
                                   SkewS1S2RiseRise;
        ELSIF (Signal2Delay > Signal1Delay) THEN
            SkewData.ExpectedType := s2r;
            TriggerDelay := SkewS1S2RiseRise +
                                     Signal2Delay - Signal1Delay;
        ELSIF (Signal1Delay > Signal2Delay) THEN
            SkewData.ExpectedType := s1r;
            TriggerDelay := SkewS2S1RiseRise +
                                     Signal1Delay - Signal2Delay;
        ELSIF (SkewS1S2RiseRise < SkewS2S1RiseRise) THEN
            SkewData.ExpectedType := s2r;
            TriggerDelay := SkewS1S2RiseRise;
        ELSE
            SkewData.ExpectedType := s1r;
            TriggerDelay := SkewS2S1RiseRise;
        END IF;
    ELSIF (Negedge(Signal1'LAST_VALUE , Signal1)) THEN
        IF ((Signal1Delay >= Signal2Delay) and (Signal2Delay >
                SkewS2S1RiseRise)) THEN
            ReportType := s1r;
            ReportTime := NOW - Signal2Delay +
                                   SkewS2S1RiseRise;
```

 Copyright © 2001 IEEE. All rights reserved.

```
            ELSIF ((Signal2Delay >= Signal1Delay) and (Signal1Delay >
                    SkewS1S2FallFall)) THEN
                ReportType := s2f;
                ReportTime := NOW - Signal1Delay +
                                        SkewS1S2FallFall;
            ELSIF (Signal2Delay > Signal1Delay) THEN
                SkewData.ExpectedType := s2f;
                TriggerDelay := SkewS1S2FallFall +
                                         Signal2Delay - Signal1Delay;
            ELSIF (Signal1Delay > Signal2Delay) THEN
                SkewData.ExpectedType := s1f;
                TriggerDelay := SkewS2S1FallFall +
                                         Signal1Delay - Signal2Delay;
            ELSIF (SkewS1S2FallFall < SkewS2S1FallFall) THEN
                SkewData.ExpectedType := s2f;
                TriggerDelay := SkewS1S2FallFall;
            ELSE
                SkewData.ExpectedType := s1f;
                TriggerDelay := SkewS2S1FallFall;
            END IF;
        END IF;
    ELSIF (Signal1'EVENT) THEN
        IF ( Signal1 /= Signal2) THEN
            IF ( Posedge( Signal1'LAST_VALUE,  Signal1)) THEN
                IF (SkewS1S2RiseRise > (Signal1Delay -
                        Signal2Delay)) THEN
                    SkewData.ExpectedType := s2r;
                    TriggerDelay := SkewS1S2RiseRise +
                                             Signal2Delay -
                                             Signal1Delay;
                ELSE
                    ReportType := s2r;
                    ReportTime := NOW + SkewS1S2RiseRise -
                                           Signal1Delay;
                END IF;
            ELSIF ( Negedge( Signal1'LAST_VALUE,  Signal1)) THEN
                IF (SkewS1S2FallFall > (Signal1Delay -
                        Signal2Delay)) THEN
                    SkewData.ExpectedType := s2f;
                    TriggerDelay := SkewS1S2FallFall +
                                             Signal2Delay -
                                             Signal1Delay;
                ELSE
                    ReportType := s2f;
                    ReportTime := NOW + SkewS1S2FallFall -
                                           Signal1Delay;
                END IF;
            END IF;
        ELSE
            IF ( Posedge( Signal1'LAST_VALUE,  Signal1)) THEN
                IF ((Signal1Delay - SkewS1S2RiseRise) >
                        (Signal2'LAST_EVENT + Signal2Delay)) THEN
                    IF ((SkewData.Signal2Old2 - Signal2Delay) >
                            (NOW - Signal1Delay +
                            SkewS1S2RiseRise)) THEN
                        ViolationCertain := FALSE;
                        ReportType := s2r;
                        ReportTime := NOW + SkewS1S2RiseRise -
                                           Signal1Delay;
                    END IF;
                END IF;
            ELSIF ( Negedge( Signal1'LAST_VALUE,  Signal1)) THEN
                IF ((Signal1Delay - SkewS1S2FallFall) >
                        (Signal2'LAST_EVENT + Signal2Delay)) THEN
                    IF (( SkewData.Signal2Old2 - Signal2Delay) >
                            (NOW - Signal1Delay +
                            SkewS1S2FallFall )) THEN
                        ViolationCertain := FALSE;
                        ReportType := s2f;
                        ReportTime := NOW + SkewS1S2FallFall -
                                           Signal1Delay;
                    END IF;
                END IF;
            END IF;
        END IF;
    ELSIF (Signal2'EVENT) THEN
        IF (Signal1 /= Signal2) THEN
            IF (Posedge(Signal2'LAST_VALUE,Signal2)) THEN
                IF ( SkewS2S1RiseRise > (Signal2Delay -
                        Signal1Delay)) THEN
                    SkewData.ExpectedType := s1r;
```


Copyright © 2001 IEEE. All rights reserved.

```
                TriggerDelay := SkewS2S1RiseRise +
                                         Signal1Delay -
                                         Signal2Delay;
            ELSE
                ReportType := s2r;
                ReportTime := NOW + SkewS2S1RiseRise -
                                       Signal2Delay;
            END IF;
        ELSIF (Negedge(Signal2'LAST_VALUE,Signal2)) THEN
            IF ( SkewS2S1FallFall > (Signal2Delay -
                    Signal1Delay)) THEN
                SkewData.ExpectedType := s1f;
                TriggerDelay := SkewS2S1FallFall +
                                         Signal1Delay -
                                         Signal2Delay;
            ELSE
                ReportType := s1f;
                ReportTime := NOW + SkewS2S1FallFall -
                                       Signal2Delay;
            END IF;
        END IF;
    ELSE
        IF (Posedge(Signal2'LAST_VALUE, Signal2)) THEN
            IF ((Signal2Delay - SkewS2S1RiseRise) >
                    (Signal1'LAST_EVENT + Signal1Delay)) THEN
                IF (( SkewData.Signal1Old2 - Signal1Delay) >
                        (NOW - Signal2Delay +
                        SkewS2S1RiseRise )) THEN
                    ViolationCertain := FALSE;
                    ReportType := s1r;
                    ReportTime := NOW + SkewS2S1RiseRise -
                                       Signal2Delay;
                END IF;
            END IF;
        ELSIF (Negedge(Signal2'LAST_VALUE, Signal2)) THEN
            IF ((Signal2Delay - SkewS2S1FallFall) >
                    (Signal1'LAST_EVENT + Signal1Delay)) THEN
                IF (( SkewData.Signal1Old2 - Signal1Delay) >
                        (NOW - Signal2Delay +
                        SkewS2S1FallFall )) THEN
                    ViolationCertain := FALSE;
                    ReportType := s1f;
                    ReportTime := NOW + SkewS2S1FallFall -
                                       Signal2Delay;
                END IF;
            END IF;
        END IF;
    END IF;
END IF;

IF (ReportType /= none) THEN
    IF (MsgOn) THEN
        CASE ReportType IS
        WHEN s1r =>
            ReportSkewViolation(
                Signal2Name,
                Signal1Name,
                SkewS2S1RiseRise,
                ReportTime,
                HeaderMsg,
                MsgSeverity,
                Inphase,
                ViolationCertain);
        WHEN s1f =>
            ReportSkewViolation(
                Signal2Name,
                Signal1Name,
                SkewS2S1FallFall,
                ReportTime,
                HeaderMsg,
                MsgSeverity,
                Inphase,
                ViolationCertain);
        WHEN s2r =>
            ReportSkewViolation(
                Signal1Name,
                Signal2Name,
                SkewS1S2RiseRise,
                ReportTime,
                HeaderMsg,
                MsgSeverity,
```

 Copyright © 2001 IEEE. All rights reserved.

```
                        Inphase,
                        ViolationCertain);
                WHEN s2f =>
                    ReportSkewViolation(
                        Signal1Name,
                        Signal2Name,
                        SkewS1S2FallFall,
                        ReportTime,
                        HeaderMsg,
                        MsgSeverity,
                        Inphase,
                        ViolationCertain);
                WHEN OTHERS =>
                END CASE;
            END IF;
            IF (XOn) THEN
                Violation := 'X';
            END IF;
            SkewData.ExpectedType := none;
        END IF;
        IF (TriggerDelay /= noTrigger) THEN
            IF (TriggerDelay = 0 ns) THEN
                trigger <= TRANSPORT trigger AFTER 0 ns;
            ELSE
                trigger <= TRANSPORT not (trigger) AFTER
                           TriggerDelay;
            END IF;
        END IF;
    END IF;
    IF (Signal1'EVENT and SkewData.Signal1Old1 /= NOW) THEN
        SkewData.Signal1Old2 := SkewData.Signal1Old1;
        SkewData.Signal1Old1 := NOW;
    END IF;
    IF (Signal2'EVENT and SkewData.Signal2Old1 /= NOW) THEN
        SkewData.Signal2Old2 := SkewData.Signal2Old1;
        SkewData.Signal2Old1 := NOW;
    END IF;
END VitalInPhaseSkewCheck;

PROCEDURE VitalOutPhaseSkewCheck (
    VARIABLE Violation         : OUT    X01;
    VARIABLE SkewData          : INOUT  VitalSkewDataType;
    SIGNAL   Signal1           : IN     std_ulogic;
    CONSTANT Signal1Name       : IN     STRING := "";
    CONSTANT Signal1Delay      : IN     TIME := 0 ns;
    SIGNAL   Signal2           : IN     std_ulogic;
    CONSTANT Signal2Name       : IN     STRING := "";
    CONSTANT Signal2Delay      : IN     TIME := 0 ns;
    CONSTANT SkewS1S2RiseFall  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS2S1RiseFall  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS1S2FallRise  : IN     TIME := TIME'HIGH;
    CONSTANT SkewS2S1FallRise  : IN     TIME := TIME'HIGH;
    CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
    CONSTANT XOn               : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT HeaderMsg         : IN     STRING  := "";
    SIGNAL   Trigger           : INOUT  std_ulogic
) IS
    VARIABLE ReportType  : VitalSkewExpectedType := none;
    VARIABLE ExpectedType  : VitalSkewExpectedType := none;
    VARIABLE ReportTime  : TIME;
    VARIABLE TriggerDelay  : TIME;
    VARIABLE ViolationCertain  : Boolean := TRUE;
BEGIN
    Violation := '0';
    TriggerDelay := noTrigger;
    IF (CheckEnabled) THEN
        IF (SkewData.ExpectedType /= none) THEN
            IF (trigger'Event) THEN
                CASE SkewData.ExpectedType IS
                WHEN s1r => ReportType := s1r;
                            ReportTime := NOW - Signal1Delay;
                WHEN s1f => ReportType := s1f;
                            ReportTime := NOW - Signal1Delay;
                WHEN s2r => ReportType := s2r;
                            ReportTime := NOW - Signal2Delay;
                WHEN s2f => ReportType := s2f;
                            ReportTime := NOW - Signal2Delay;
                WHEN OTHERS =>
                END CASE;
```

Copyright © 2001 IEEE. All rights reserved.

```
            SkewData.ExpectedType := none;
        ELSIF (Signal1'Event OR Signal2'Event ) THEN
            IF (Signal1 /= 'X' AND Signal2 /= 'X' ) THEN
                TriggerDelay := 0 ns;
                SkewData.ExpectedType := none;
            END IF;
        END IF;
    END IF;

    IF (Signal1'EVENT and Signal2'EVENT) THEN
        IF (Signal1 /= Signal2) THEN
            IF (Posedge(Signal1'LAST_VALUE, Signal1)) THEN
                IF ((Signal1Delay - Signal2Delay) >=
                                SkewS1S2RiseFall) THEN
                    ReportType := s2f;
                    ReportTime := NOW - Signal1Delay +
                                  SkewS1S2RiseFall;
                ELSIF ((Signal2Delay - Signal1Delay) >=
                                SkewS2S1FallRise) THEN
                    ReportType := s1r;
                    ReportTime := NOW - Signal2Delay +
                                  SkewS2S1FallRise;
                END IF;
            ELSIF (Negedge(Signal1'LAST_VALUE, Signal1)) THEN
                IF ((Signal1Delay - Signal2Delay) >=
                        SkewS1S2FallRise) THEN
                    ReportType := s2r;
                    ReportTime := NOW - Signal1Delay +
                                  SkewS1S2FallRise;
                ELSIF ((Signal2Delay - Signal1Delay) >=
                        SkewS2S1RiseFall) THEN
                    ReportType := s1f;
                    ReportTime := NOW - Signal2Delay +
                                  SkewS2S1RiseFall;
                END IF;
            END IF;
        ELSIF (Posedge(Signal1'LAST_VALUE, Signal1)) THEN
            IF ((Signal1Delay >= Signal2Delay) and (Signal2Delay >
                      SkewS2S1RiseFall)) THEN
                ReportType := s1f;
                ReportTime := NOW - Signal2Delay +
                              SkewS2S1RiseFall;
            ELSIF ((Signal2Delay >= Signal1Delay) and (Signal1Delay >
                              SkewS1S2RiseFall)) THEN
                ReportType := s2f;
                ReportTime := NOW - Signal1Delay +
                              SkewS1S2RiseFall;
            ELSIF (Signal1Delay > Signal2Delay) THEN
                SkewData.ExpectedType := s1f;
                TriggerDelay := SkewS2S1RiseFall +
                                Signal1Delay - Signal2Delay;
            ELSIF (Signal2Delay > Signal1Delay) THEN
                SkewData.ExpectedType := s2f;
                TriggerDelay := SkewS1S2RiseFall +
                                Signal2Delay - Signal1Delay;
            ELSIF (SkewS2S1RiseFall < SkewS1S2RiseFall) THEN
                SkewData.ExpectedType := s1f;
                TriggerDelay := SkewS2S1RiseFall;
            ELSE
                SkewData.ExpectedType := s2f;
                TriggerDelay := SkewS1S2RiseFall;
            END IF;
        ELSIF (Negedge(Signal1'LAST_VALUE, Signal1)) THEN
            IF ((Signal1Delay >= Signal2Delay) and (Signal2Delay >
                                        SkewS2S1FallRise)) THEN
                ReportType := s1r;
                ReportTime := NOW - Signal2Delay +
                              SkewS2S1FallRise;
            ELSIF ((Signal2Delay >= Signal1Delay) and (Signal1Delay >
                                    SkewS1S2FallRise)) THEN
                ReportType := s2r;
                ReportTime := NOW - Signal1Delay +
                              SkewS1S2FallRise;
            ELSIF (Signal1Delay > Signal2Delay) THEN
                SkewData.ExpectedType := s1r;
                TriggerDelay := SkewS2S1FallRise +
                                Signal1Delay - Signal2Delay;
            ELSIF (Signal2Delay > Signal1Delay) THEN
                SkewData.ExpectedType := s2r;
                TriggerDelay := SkewS1S2FallRise +
                                        Signal2Delay - Signal1Delay;
```

 Copyright © 2001 IEEE. All rights reserved.

```
            ELSIF (SkewS2S1FallRise < SkewS1S2FallRise) THEN
                SkewData.ExpectedType := s1r;
                TriggerDelay := SkewS2S1FallRise;
            ELSE
                SkewData.ExpectedType := s2r;
                TriggerDelay := SkewS1S2FallRise;
            END IF;
        END IF;
    ELSIF (Signal1'EVENT) THEN
        IF (Signal1 = Signal2) THEN
            IF (Posedge(Signal1'LAST_VALUE,Signal1)) THEN
                IF (SkewS1S2RiseFall > (Signal1Delay -
                                           Signal2Delay)) THEN
                    SkewData.ExpectedType := s2f;
                    TriggerDelay := SkewS1S2RiseFall +
                                    Signal2Delay - Signal1Delay;
                ELSE
                    ReportType := s2f;
                    ReportTime := NOW - Signal1Delay +
                                  SkewS1S2RiseFall;
                END IF;
            ELSIF ( Negedge(Signal1'LAST_VALUE, Signal1)) THEN
                IF ( SkewS1S2FallRise > (Signal1Delay -
                                           Signal2Delay)) THEN
                    SkewData.ExpectedType := s2r;
                    TriggerDelay := SkewS1S2FallRise +
                                    Signal2Delay - Signal1Delay;
                ELSE
                    ReportType := s2r;
                    ReportTime := NOW - Signal1Delay +
                                  SkewS1S2FallRise;
                END IF;
            END IF;
        ELSE
            IF (Posedge( Signal1'LAST_VALUE, Signal1 )) THEN
                IF ((Signal1Delay - SkewS1S2RiseFall) >
                    (Signal2'LAST_EVENT + Signal2Delay)) THEN
                    IF (( SkewData.Signal2Old2 - Signal2Delay) >
                            (NOW - Signal1Delay +
                            SkewS1S2RiseFall )) THEN
                        ViolationCertain := FALSE;
                        ReportType := s2f;
                        ReportTime := NOW + SkewS1S2RiseFall -
                                           Signal1Delay;
                    END IF;
                END IF;
            ELSIF (Negedge(Signal1'LAST_VALUE, Signal1)) THEN
                IF ((Signal1Delay - SkewS1S2FallRise) >
                    (Signal2'LAST_EVENT + Signal2Delay)) THEN
                    IF (( SkewData.Signal2Old2 - Signal2Delay) >
                            (NOW - Signal1Delay +
                            SkewS1S2FallRise )) THEN
                        ViolationCertain := FALSE;
                        ReportType := s2r;
                        ReportTime := NOW + SkewS1S2FallRise -
                                           Signal1Delay;
                    END IF;
                END IF;
            END IF;
        END IF;
    ELSIF (Signal2'EVENT) THEN
        IF (Signal1 = Signal2) THEN
            IF (Posedge(Signal2'LAST_VALUE,Signal2)) THEN
                IF (SkewS2S1RiseFall > (Signal2Delay -
                                           Signal1Delay)) THEN
                    SkewData.ExpectedType := s1f;
                    TriggerDelay := SkewS2S1RiseFall + Signal1Delay -
                                          Signal2Delay ;
                ELSE
                    ReportType := s1f;
                    ReportTime := NOW + SkewS2S1RiseFall -
                                        Signal2Delay;
                END IF;
            ELSIF (Negedge(Signal2'LAST_VALUE,Signal2)) THEN
                IF (SkewS2S1FallRise > (Signal2Delay -
                                          Signal1Delay)) THEN
                    SkewData.ExpectedType := s1r;
                    TriggerDelay := SkewS2S1FallRise + Signal1Delay -
                                          Signal2Delay;
                ELSE
                    ReportType := s1r;
```

Copyright © 2001 IEEE. All rights reserved.

```
                ReportTime := NOW + SkewS2S1FallRise -
                                    Signal2Delay;
            END IF;
        END IF;
    ELSE
        IF (Posedge(Signal2'LAST_VALUE,Signal2)) THEN
            IF ((Signal2Delay - SkewS2S1RiseFall) >
                  (Signal1'LAST_EVENT + Signal1Delay)) THEN
                IF (( SkewData.Signal1Old2 - Signal1Delay) >
                        (NOW - Signal2Delay +
                        SkewS2S1RiseFall )) THEN
                    ViolationCertain := FALSE;
                    ReportType := s1f;
                    ReportTime := NOW + SkewS2S1RiseFall -
                                        Signal2Delay;
                END IF;
            END IF;
        ELSIF (Negedge(Signal2'LAST_VALUE,Signal2)) THEN
            IF ((Signal2Delay - SkewS2S1FallRise) >
                   (Signal1'LAST_EVENT + Signal1Delay)) THEN
                IF (( SkewData.Signal1Old2 - Signal1Delay) >
                        (NOW - Signal2Delay +
                        SkewS2S1FallRise )) THEN
                    ViolationCertain := FALSE;
                    ReportType := s1r;
                    ReportTime := NOW + SkewS2S1FallRise -
                                        Signal2Delay;
                END IF;
            END IF;
        END IF;
    END IF;
END IF;

IF (ReportType /= none) THEN
    IF (MsgOn) THEN
        CASE ReportType IS
        WHEN s1r =>
            ReportSkewViolation(
                Signal2Name,
                Signal1Name,
                SkewS2S1FallRise,
                ReportTime,
                HeaderMsg,
                MsgSeverity,
                Outphase,
                ViolationCertain);
        WHEN s1f =>
            ReportSkewViolation(
                Signal2Name,
                Signal1Name,
                SkewS2S1RiseFall,
                ReportTime,
                HeaderMsg,
                MsgSeverity,
                Outphase,
                ViolationCertain);
        WHEN s2r =>
            ReportSkewViolation(
                Signal1Name,
                Signal2Name,
                SkewS1S2FallRise,
                ReportTime,
                HeaderMsg,
                MsgSeverity,
                Outphase,
                ViolationCertain);
        WHEN s2f =>
            ReportSkewViolation(
                Signal1Name,
                Signal2Name,
                SkewS1S2RiseFall,
                ReportTime,
                HeaderMsg,
                MsgSeverity,
                Outphase,
                ViolationCertain);
        WHEN OTHERS =>
        END CASE;
    END IF;
    IF (XOn) THEN
        Violation := 'X';
```

 Copyright © 2001 IEEE. All rights reserved.

```
                END IF;
                ReportType := none;
            END IF;
            IF (TriggerDelay /= noTrigger) THEN
                IF (TriggerDelay = 0 ns) THEN
                    trigger <= TRANSPORT trigger AFTER 0 ns;
                ELSE
                    trigger <= TRANSPORT not (trigger) AFTER
                               TriggerDelay;
                END IF;
            END IF;
        END IF;
        IF (Signal1'EVENT and SkewData.Signal1Old1 /= NOW) THEN
            SkewData.Signal1Old2 := SkewData.Signal1Old1;
            SkewData.Signal1Old1 := NOW;
        END IF;
        IF (Signal2'EVENT and SkewData.Signal2Old1 /= NOW) THEN
            SkewData.Signal2Old2 := SkewData.Signal2Old1;
            SkewData.Signal2Old1 := NOW;
        END IF;
    END VitalOutPhaseSkewCheck;

END VITAL_Timing;
```

13.3 VITAL_Primitives package declaration

```
--------------------------------------------------------------------------------
-- Title         : Standard VITAL_Primitives Package
--               : $Revision: 1.2 $
--               :
-- Library       : VITAL
--               :
-- Developers    : IEEE DASC Timing Working Group (TWG), PAR 1076.4
--               :
-- Purpose       : This packages defines standard types, constants, functions
--               : and procedures for use in developing ASIC models.
--               : Specifically a set of logic primitives are defined.
--               :
-- -----------------------------------------------------------------------------
--
-- -----------------------------------------------------------------------------
-- Modification History :
-- -----------------------------------------------------------------------------
-- Version No:|Auth:| Mod.Date:| Changes Made:
--   v95.0 A  |     | 06/02/95 | Initial ballot draft 1995
--   v95.1    |     | 08/31/95 | #204 - glitch detection prior to OutputMap
-- -----------------------------------------------------------------------------
--   v95.2    | ddl | 09/14/96 | #223 - single input prmtvs use on-detect
--            |     |          |        instead of glitch-on-event behavior
--   v95.3    | ddl | 09/24/96 | #236 - VitalTruthTable DataIn should be of
--            |     |          |        of class SIGNAL
--   v95.4    | ddl | 01/16/97 | #243 - index constraint error in nbit xor/xnor
--   v99.1    | dbb | 03/31/99 | Updated for VHDL 93
-- -----------------------------------------------------------------------------

LIBRARY STD;
USE STD.TEXTIO.ALL;

PACKAGE BODY VITAL_Primitives IS
    -- -------------------------------------------------------------------------
    --  Default values for Primitives
    -- -------------------------------------------------------------------------
    --  default values for delay parameters
    CONSTANT VitalDefDelay01  : VitalDelayType01  := VitalZeroDelay01;
    CONSTANT VitalDefDelay01Z : VitalDelayType01Z := VitalZeroDelay01Z;

    TYPE VitalTimeArray IS ARRAY (NATURAL RANGE <>) OF TIME;

    --  default primitive model operation parameters
    --  Glitch detection/reporting
    TYPE VitalGlitchModeType IS ( MessagePlusX, MessageOnly, XOnly, NoGlitch);
    CONSTANT PrimGlitchMode : VitalGlitchModeType   := XOnly;

    -- -------------------------------------------------------------------------
    -- Local Type and Subtype Declarations
    -- -------------------------------------------------------------------------
    ---------------------------------------------------------------------------
    -- enumeration value representing the transition or level of the signal.
```

Copyright © 2001 IEEE. All rights reserved.

```
--  See function 'GetEdge'
---------------------------------------------------------------------------
TYPE EdgeType IS ( 'U',   -- Uninitialized level
                   'X',   -- Unknown level
                   '0',   -- low level
                   '1',   -- high level
                   '\',   -- 1 to 0 falling edge
                   '/',   -- 0 to 1 rising  edge
                   'F',   -- * to 0 falling edge
                   'R',   -- * to 1 rising  edge
                   'f',   -- rising  to X edge
                   'r',   -- falling to X edge
                   'x',   -- Unknown edge (ie U->X)
                   'V'    -- Timing violation edge
                 );
TYPE EdgeArray  IS ARRAY ( NATURAL RANGE <> ) OF EdgeType;

TYPE EdgeX1Table IS ARRAY ( EdgeType                          ) OF EdgeType;
TYPE EdgeX2Table IS ARRAY ( EdgeType, EdgeType                ) OF EdgeType;
TYPE EdgeX3Table IS ARRAY ( EdgeType, EdgeType, EdgeType      ) OF EdgeType;
TYPE EdgeX4Table IS ARRAY (EdgeType,EdgeType,EdgeType,EdgeType) OF EdgeType;

TYPE LogicToEdgeT  IS ARRAY(std_ulogic, std_ulogic) OF EdgeType;
TYPE LogicToLevelT IS ARRAY(std_ulogic ) OF EdgeType;

TYPE GlitchDataType IS
  RECORD
    SchedTime    : TIME;
    GlitchTime   : TIME;
    SchedValue   : std_ulogic;
    CurrentValue    : std_ulogic;
  END RECORD;
TYPE GlitchDataArrayType IS ARRAY (NATURAL RANGE <>)
     OF GlitchDataType;

-- Enumerated type used in selection of output path delays
TYPE SchedType  IS
  RECORD
    inp0  : TIME;   -- time (abs) of output change due to input change to 0
    inp1  : TIME;   -- time (abs) of output change due to input change to 1
    InpX  : TIME;   -- time (abs) of output change due to input change to X
    Glch0 : TIME;   -- time (abs) of output glitch due to input change to 0
    Glch1 : TIME;   -- time (abs) of output glitch due to input change to 0
  END RECORD;

TYPE SchedArray  IS ARRAY ( NATURAL RANGE <> ) OF SchedType;
CONSTANT DefSchedType : SchedType := (TIME'HIGH, TIME'HIGH, 0 ns,0 ns,0 ns);
CONSTANT DefSchedAnd  : SchedType := (TIME'HIGH, 0 ns,0 ns, TIME'HIGH,0 ns);

-- Constrained array declarations (common sizes used by primitives)
SUBTYPE SchedArray2 IS SchedArray(1 DOWNTO 0);
SUBTYPE SchedArray3 IS SchedArray(2 DOWNTO 0);
SUBTYPE SchedArray4 IS SchedArray(3 DOWNTO 0);
SUBTYPE SchedArray8 IS SchedArray(7 DOWNTO 0);

SUBTYPE TimeArray2 IS VitalTimeArray(1 DOWNTO 0);
SUBTYPE TimeArray3 IS VitalTimeArray(2 DOWNTO 0);
SUBTYPE TimeArray4 IS VitalTimeArray(3 DOWNTO 0);
SUBTYPE TimeArray8 IS VitalTimeArray(7 DOWNTO 0);

SUBTYPE GlitchArray2 IS GlitchDataArrayType(1 DOWNTO 0);
SUBTYPE GlitchArray3 IS GlitchDataArrayType(2 DOWNTO 0);
SUBTYPE GlitchArray4 IS GlitchDataArrayType(3 DOWNTO 0);
SUBTYPE GlitchArray8 IS GlitchDataArrayType(7 DOWNTO 0);

SUBTYPE EdgeArray2 IS EdgeArray(1 DOWNTO 0);
SUBTYPE EdgeArray3 IS EdgeArray(2 DOWNTO 0);
SUBTYPE EdgeArray4 IS EdgeArray(3 DOWNTO 0);
SUBTYPE EdgeArray8 IS EdgeArray(7 DOWNTO 0);

CONSTANT DefSchedArray2 : SchedArray2 :=
                         (OTHERS=> (0 ns, 0 ns, 0 ns, 0 ns, 0 ns));

TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

CONSTANT InitialEdge : LogicToLevelT := (
        '1'|'H'  => 'R',
        '0'|'L'  => 'F',
        OTHERS   => 'x'
 );
```

 Copyright © 2001 IEEE. All rights reserved.

```
CONSTANT LogicToEdge  : LogicToEdgeT  := (  -- previous, current
--  old \ new: U     X     0     1     Z     W     L     H     -
    'U'  =>  ( 'U' , 'x' , 'F' , 'R' , 'x' , 'x' , 'F' , 'R' , 'x'  ),
    'X'  =>  ( 'x' , 'X' , 'F' , 'R' , 'x' , 'X' , 'F' , 'R' , 'X'  ),
    '0'  =>  ( 'r' , 'r' , '0' , '/' , 'r' , 'r' , '0' , '/' , 'r'  ),
    '1'  =>  ( 'f' , 'f' , '\' , '1' , 'f' , 'f' , '\' , '1' , 'f'  ),
    'Z'  =>  ( 'x' , 'X' , 'F' , 'R' , 'X' , 'x' , 'F' , 'R' , 'x'  ),
    'W'  =>  ( 'x' , 'X' , 'F' , 'R' , 'x' , 'X' , 'F' , 'R' , 'X'  ),
    'L'  =>  ( 'r' , 'r' , '0' , '/' , 'r' , 'r' , '0' , '/' , 'r'  ),
    'H'  =>  ( 'f' , 'f' , '\' , '1' , 'f' , 'f' , '\' , '1' , 'f'  ),
    '-'  =>  ( 'x' , 'X' , 'F' , 'R' , 'x' , 'X' , 'F' , 'R' , 'X'  )
);
CONSTANT LogicToLevel : LogicToLevelT := (
        '1' |' H'  => '1' ,
        '0' |' L'  => '0' ,
        'U'        => 'U' ,
        OTHERS     => 'X'
 );

-- ----------------------------------
-- 3-state logic tables
-- ----------------------------------
CONSTANT BufIf0_Table : stdlogic_table :=
    -- enable        data       value
    ( '1' |' H'   => ( OTHERS  => 'Z' ),
      '0' |' L'   => ( '1' |' H'  => '1' ,
                       '0' |' L'  => '0' ,
                       'U'        => 'U' ,
                       OTHERS  => 'X' ),
      'U'         => ( OTHERS  => 'U' ),
      OTHERS      => ( OTHERS  => 'X' ) );
CONSTANT BufIf1_Table : stdlogic_table :=
    -- enable        data       value
    ( '0' |' L'   => ( OTHERS  => 'Z' ),
      '1' |' H'   => ( '1' |' H'  => '1' ,
                       '0' |' L'  => '0' ,
                       'U'        => 'U' ,
                       OTHERS  => 'X' ),
      'U'         => ( OTHERS  => 'U' ),
      OTHERS      => ( OTHERS  => 'X' ) );
CONSTANT InvIf0_Table : stdlogic_table :=
    -- enable        data       value
    ( '1' |' H'   => ( OTHERS  => 'Z' ),
      '0' |' L'   => ( '1' |' H'  => '0' ,
                       '0' |' L'  => '1' ,
                       'U'        => 'U' ,
                       OTHERS  => 'X' ),
      'U'         => ( OTHERS  => 'U' ),
      OTHERS      => ( OTHERS  => 'X' ) );
CONSTANT InvIf1_Table : stdlogic_table :=
    -- enable        data       value
    ( '0' |' L'   => ( OTHERS  => 'Z' ),
      '1' |' H'   => ( '1' |' H'  => '0' ,
                       '0' |' L'  => '1' ,
                       'U'        => 'U' ,
                       OTHERS  => 'X' ),
      'U'         => ( OTHERS  => 'U' ),
      OTHERS      => ( OTHERS  => 'X' ) );


TYPE To_StateCharType IS ARRAY (VitalStateSymbolType) OF CHARACTER;
CONSTANT To_StateChar : To_StateCharType :=
 ( '/' , '\' , 'P' , 'N' , 'r' , 'f' , 'p' , 'n' , 'R' , 'F' , '^' , 'v' ,
   'E' , 'A' , 'D' , '*' , 'X' , '0' , '1' , '-' , 'B' , 'Z' , 'S' );
TYPE To_TruthCharType IS ARRAY (VitalTruthSymbolType) OF CHARACTER;
CONSTANT To_TruthChar : To_TruthCharType :=
 ( 'X' , '0' , '1' , '-' , 'B' , 'Z' );

TYPE TruthTableOutMapType IS ARRAY (VitalTruthSymbolType) OF std_ulogic;
CONSTANT TruthTableOutMap : TruthTableOutMapType :=
   --  'X' , '0' , '1' , '-' , 'B' , 'Z'
     ( 'X' , '0' , '1' , 'X' , '-' , 'Z' );

TYPE StateTableOutMapType IS ARRAY (VitalStateSymbolType) OF std_ulogic;
-- does conversion to X01Z or '-' if invalid
CONSTANT StateTableOutMap : StateTableOutMapType :=
 -- '/'  '\'  'P'  'N'  'r'  'f'  'p'  'n'  'R'  'F'  '^'  'v'
 -- 'E'  'A'  'D'  '*'  'X'  '0'  '1'  '-'  'B'  'Z'  'S'
  ( '-' ,' -' ,' -' ,' -' ,' -' ,' -' ,' -' ,' -' ,' -' ,' -' ,' -' ,' -' ,
    '-' ,' -' ,' -' ,' -' ,' X' ,' 0' ,' 1' ,' X' ,' -' ,' Z' ,' W' );
```

Copyright © 2001 IEEE. All rights reserved.

```
-- -------------------------------------------------------------------------
TYPE ValidTruthTableInputType IS ARRAY (VitalTruthSymbolType) OF BOOLEAN;
-- checks if a symbol IS valid for the stimulus portion of a truth table
CONSTANT ValidTruthTableInput : ValidTruthTableInputType :=
   -- 'X'     '0'     '1'     '-'     'B'     'Z'
   (  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  FALSE );

TYPE TruthTableMatchType IS ARRAY (X01, VitalTruthSymbolType) OF BOOLEAN;
-- checks if an input matches th corresponding truth table symbol
-- use: TruthTableMatch(input_converted_to_X01, truth_table_stimulus_symbol)
CONSTANT TruthTableMatch : TruthTableMatchType  :=  (
   -- X,      0,      1,      -       B       Z
   (  TRUE,  FALSE, FALSE, TRUE,  FALSE, FALSE ),  -- X
   (  FALSE, TRUE,  FALSE, TRUE,  TRUE,  FALSE ),  -- 0
   (  FALSE, FALSE, TRUE,  TRUE,  TRUE,  FALSE )   -- 1
);

-- -------------------------------------------------------------------------
TYPE ValidStateTableInputType IS ARRAY (VitalStateSymbolType) OF BOOLEAN;
CONSTANT ValidStateTableInput : ValidStateTableInputType :=
   -- '/',    '\',    'P',    'N',    'r',    'f',
  (   TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,
   -- 'p',    'n',    'R',    'F',    '^',    'v',
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,
   -- 'E',    'A',    'D',    '*',
      TRUE,  TRUE,  TRUE,  TRUE,
   -- 'X',    '0',    '1',    '-',    'B',    'Z',
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE, FALSE,
   -- 'S'
      TRUE );

CONSTANT ValidStateTableState : ValidStateTableInputType :=
   -- '/',    '\',    'P',    'N',    'r',    'f',
  (   FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
   -- 'p',    'n',    'R',    'F',    '^',    'v',
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
   -- 'E',    'A',    'D',    '*',
      FALSE, FALSE, FALSE, FALSE,
   -- 'X',    '0',    '1',    '-',    'B',    'Z',
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  FALSE,
   -- 'S'
      FALSE );

TYPE StateTableMatchType IS ARRAY (X01,X01,VitalStateSymbolType) OF BOOLEAN;
-- last value, present value, table symbol
CONSTANT StateTableMatch : StateTableMatchType :=  (
  ( -- X (lastvalue)
 -- /     \     P     N     r     f
 -- p     n     R     F     ^     v
 -- E     A     D     *
 -- X     0     1     -     B     Z     S
  (FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,
   TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE),
  (FALSE,FALSE,FALSE,TRUE, FALSE,FALSE,
   FALSE,FALSE,FALSE,TRUE, FALSE,TRUE,
   TRUE, FALSE,TRUE, TRUE,
   FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,FALSE),
  (FALSE,FALSE,TRUE, FALSE,FALSE,FALSE,
   FALSE,FALSE,TRUE, FALSE,TRUE, FALSE,
   TRUE, TRUE, FALSE,TRUE,
   FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,FALSE)
  ),

  (-- 0 (lastvalue)
 -- /     \     P     N     r     f
 -- p     n     R     F     ^     v
 -- E     A     D     *
 -- X     0     1     -     B     Z     S
  (FALSE,FALSE,FALSE,FALSE,TRUE, FALSE,
   TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
   FALSE,TRUE, FALSE,TRUE,
   TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE),
  (FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,
   FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,TRUE ),
  (TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
   TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,TRUE,
```

 Copyright © 2001 IEEE. All rights reserved.

```
   FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,FALSE)
  ),

  (-- 1 (lastvalue)
 -- /     \     P     N     r     f
 -- p     n     R     F     ^     v
 -- E     A     D     *
 -- X     0     1     -     B     Z     S
  (FALSE,FALSE,FALSE,FALSE,FALSE,TRUE ,
   FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
   FALSE,FALSE,TRUE, TRUE,
   TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE),
  (FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
   FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
   FALSE,FALSE,FALSE,TRUE,
   FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,FALSE),
  (FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,TRUE )
  )
  );

TYPE Logic_UX01Z_Table IS ARRAY (std_ulogic) OF UX01Z;
----------------------------------------------------------
-- table name : cvt_to_x01z
-- parameters :  std_ulogic  -- some logic value
-- returns    :  UX01Z       -- state value of logic value
-- purpose    :  to convert state-strength to state only
----------------------------------------------------------
CONSTANT cvt_to_ux01z : Logic_UX01Z_Table :=
                            ('U','X','0','1','Z','X','0','1','X' );

TYPE LogicCvtTableType IS ARRAY (std_ulogic) OF CHARACTER;
CONSTANT LogicCvtTable : LogicCvtTableType
                 := ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

--------------------------------------------------------------------
-- LOCAL Utilities
--------------------------------------------------------------------
-- -------------------------------------------------------------------------
--  FUNCTION  NAME  :  MINIMUM
--
--  PARAMETERS      :  in1, in2  - integer, time
--
--  DESCRIPTION     :  return smaller of in1 and in2
-- -------------------------------------------------------------------------
FUNCTION Minimum (
        CONSTANT in1, in2 : INTEGER
      ) RETURN INTEGER IS
BEGIN
   IF (in1 < in2) THEN
      RETURN in1;
   END IF;
   RETURN in2;
END;
-- -------------------------------------------------------------------------
FUNCTION Minimum (
        CONSTANT t1,t2 : IN TIME
      ) RETURN TIME IS
BEGIN
    IF ( t1 < t2 ) THEN RETURN (t1); ELSE RETURN (t2); END IF;
END Minimum;

-- -------------------------------------------------------------------------
--  FUNCTION  NAME  :  MAXIMUM
--
--  PARAMETERS      :  in1, in2  - integer, time
--
--  DESCRIPTION     :  return larger of in1 and in2
-- -------------------------------------------------------------------------
FUNCTION Maximum (
        CONSTANT in1, in2 : INTEGER
      ) RETURN INTEGER IS
BEGIN
   IF (in1 > in2) THEN
      RETURN in1;
   END IF;
   RETURN in2;
END;
-----------------------------------------------------------------------
```

Copyright © 2001 IEEE. All rights reserved.

```
FUNCTION Maximum (
        CONSTANT t1,t2 : IN TIME
      ) RETURN TIME IS
BEGIN
    IF ( t1 > t2 ) THEN RETURN (t1); ELSE RETURN (t2); END IF;
END Maximum;

-----------------------------------------------------------------------
IMPURE FUNCTION GlitchMinTime (
        CONSTANT Time1, Time2 : IN TIME
      ) RETURN TIME IS
BEGIN
    IF ( Time1 >= NOW ) THEN
            IF ( Time2 >= NOW ) THEN
              RETURN Minimum ( Time1, Time2);
            ELSE
              RETURN Time1;
            END IF;
    ELSE
            IF ( Time2 >= NOW ) THEN
               RETURN Time2;
            ELSE
               RETURN 0 ns;
            END IF;
    END IF;
END;

--------------------------------------------------------------------
-- Error Message Types and Tables
--------------------------------------------------------------------
TYPE VitalErrorType IS (
    ErrNegDel,
    ErrInpSym,
    ErrOutSym,
    ErrStaSym,
    ErrVctLng,
    ErrTabWidSml,
    ErrTabWidLrg,
    ErrTabResSml,
    ErrTabResLrg
);

TYPE VitalErrorSeverityType IS ARRAY (VitalErrorType) OF SEVERITY_LEVEL;
CONSTANT VitalErrorSeverity : VitalErrorSeverityType := (
    ErrNegDel    => WARNING,
    ErrInpSym    => ERROR,
    ErrOutSym    => ERROR,
    ErrStaSym    => ERROR,
    ErrVctLng    => ERROR,
    ErrTabWidSml => ERROR,
    ErrTabWidLrg => WARNING,
    ErrTabResSml => WARNING,
    ErrTabResLrg => WARNING
);

CONSTANT MsgNegDel : STRING :=
  "Negative delay. New output value not scheduled. Output signal is: ";
CONSTANT MsgInpSym : STRING :=
  "Illegal symbol in the input portion of a Truth/State table.";
CONSTANT MsgOutSym : STRING :=
  "Illegal symbol in the output portion of a Truth/State table.";
CONSTANT MsgStaSym : STRING :=
  "Illegal symbol in the state portion of a State table.";
CONSTANT MsgVctLng : STRING :=
  "Vector (array) lengths not equal. ";
CONSTANT MsgTabWidSml : STRING :=
  "Width of the Truth/State table is too small.";
CONSTANT MsgTabWidLrg : STRING :=
  "Width of Truth/State table is too large. Extra elements are ignored.";
CONSTANT MsgTabResSml : STRING :=
  "Result of Truth/State table has too many elements.";
CONSTANT MsgTabResLrg : STRING :=
  "Result of Truth/State table has too few elements.";

CONSTANT MsgUnknown : STRING :=
  "Unknown error message.";

--------------------------------------------------------------------
-- LOCAL Utilities
--------------------------------------------------------------------
FUNCTION VitalMessage (
```

 Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT ErrorId : IN VitalErrorType
      ) RETURN STRING IS
BEGIN
    CASE ErrorId IS
        WHEN ErrNegDel    => RETURN MsgNegDel;
        WHEN ErrInpSym    => RETURN MsgInpSym;
        WHEN ErrOutSym    => RETURN MsgOutSym;
        WHEN ErrStaSym    => RETURN MsgStaSym;
        WHEN ErrVctLng    => RETURN MsgVctLng;
        WHEN ErrTabWidSml => RETURN MsgTabWidSml;
        WHEN ErrTabWidLrg => RETURN MsgTabWidLrg;
        WHEN ErrTabResSml => RETURN MsgTabResSml;
        WHEN ErrTabResLrg => RETURN MsgTabResLrg;
        WHEN OTHERS       => RETURN MsgUnknown;
    END CASE;
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId)
      SEVERITY VitalErrorSeverity(ErrorId);
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType;
        CONSTANT Info    : IN STRING
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId) & Info
      SEVERITY VitalErrorSeverity(ErrorId);
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType;
        CONSTANT Info    : IN CHARACTER
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId) & Info
      SEVERITY VitalErrorSeverity(ErrorId);
END;

---------------------------------------------------------------------------
PROCEDURE ReportGlitch (
        CONSTANT GlitchRoutine  : IN  STRING;
        CONSTANT OutSignalName  : IN  STRING;
        CONSTANT PreemptedTime  : IN  TIME;
        CONSTANT PreemptedValue : IN  std_ulogic;
        CONSTANT NewTime        : IN  TIME;
        CONSTANT NewValue       : IN  std_ulogic;
        CONSTANT Index          : IN  INTEGER := 0;
        CONSTANT IsArraySignal  : IN  BOOLEAN := FALSE;
        CONSTANT MsgSeverity    : IN  SEVERITY_LEVEL := WARNING
) IS

    VARIABLE StrPtr1, StrPtr2, StrPtr3, StrPtr4, StrPtr5 : LINE;
BEGIN

    Write (StrPtr1, PreemptedTime );
    Write (StrPtr2, NewTime);
    Write (StrPtr3, LogicCvtTable(PreemptedValue));
    Write (StrPtr4, LogicCvtTable(NewValue));
    IF IsArraySignal THEN
        Write (StrPtr5, STRING'( "(" ) );
        Write (StrPtr5, Index);
        Write (StrPtr5, STRING'( ")" ) );
    ELSE
        Write (StrPtr5, STRING'( " " ) );
    END IF;

    -- Issue Report only if Preemted value has not been
    --  removed from event queue
    ASSERT PreemptedTime >  NewTime
      REPORT GlitchRoutine & ": GLITCH Detected on port " &
```

Copyright © 2001 IEEE. All rights reserved.

```
              OutSignalName & StrPtr5.ALL &
              "; Preempted Future Value := " & StrPtr3.ALL &
              " @ " & StrPtr1.ALL &
              "; Newly Scheduled Value := " & StrPtr4.ALL &
              " @ " & StrPtr2.ALL &
              ";"
       SEVERITY MsgSeverity;

    DEALLOCATE(StrPtr1);
    DEALLOCATE(StrPtr2);
    DEALLOCATE(StrPtr3);
    DEALLOCATE(StrPtr4);
    DEALLOCATE(StrPtr5);
    RETURN;
END ReportGlitch;

---------------------------------------------------------------------------
-- Procedure  : VitalGlitchOnEvent
--            :
-- Parameters : OutSignal ........ signal being driven
--            : OutSignalName..... name of the driven signal
--            : GlitchData........ internal data required by the procedure
--            : NewValue.......... new value being assigned
--            : NewDelay.......... Delay accompanying the assignment
--            :                    (Note: for vectors, this is an array)
--            : GlitchMode........ Glitch generation mode
--            :                     MessagePlusX, MessageOnly,
--            :                     XOnly, NoGlitch )
--            : GlitchDelay....... if <= 0 ns , then there will be no Glitch
--            :                    if >  NewDelay, then there is no Glitch,
--            :                    otherwise, this is the time when a FORCED
--            :                    generation of a glitch will occur.
----------------------------------------------------------------------------
PROCEDURE VitalGlitchOnEvent (
        SIGNAL   OutSignal        : OUT   std_logic;
        CONSTANT OutSignalName    : IN    STRING;
        VARIABLE GlitchData       : INOUT GlitchDataType;
        CONSTANT NewValue         : IN    std_logic;
        CONSTANT NewDelay         : IN    TIME := 0 ns;
        CONSTANT GlitchMode       : IN    VitalGlitchModeType := MessagePlusX;
        CONSTANT GlitchDelay      : IN    TIME := -1 ns;  -- IR#223
        CONSTANT MsgSeverity      : IN    SEVERITY_LEVEL := WARNING
) IS
-- ------------------------------------------------------------------------
    VARIABLE NoGlitchDet  : BOOLEAN := FALSE;
    VARIABLE OldGlitch    : BOOLEAN := FALSE;
    VARIABLE Dly          : TIME    := NewDelay;

BEGIN
    -- If nothing to schedule, just return
    IF NewDelay < 0 ns THEN
        IF (NewValue /= GlitchData.SchedValue) THEN
            VitalError ( "VitalGlitchOnEvent", ErrNegDel, OutSignalName );
        END IF;

    ELSE
        -- If nothing currently scheduled
        IF GlitchData.SchedTime <= NOW THEN
            GlitchData.CurrentValue := GlitchData.SchedValue;
            IF (GlitchDelay <= 0 ns) THEN
                IF (NewValue = GlitchData.SchedValue) THEN RETURN; END IF;
                NoGlitchDet := TRUE;
            END IF;

        -- Transaction currently scheduled - if glitch already happened
        ELSIF GlitchData.GlitchTime <= NOW THEN
            GlitchData.CurrentValue := 'X';
            OldGlitch := TRUE;
            IF (GlitchData.SchedValue = NewValue) THEN
                dly := Minimum( GlitchData.SchedTime-NOW, NewDelay );
            END IF;

        -- Transaction currently scheduled (no glitch if same value)
        ELSIF (GlitchData.SchedValue = NewValue) AND
              (GlitchData.SchedTime = GlitchData.GlitchTime) AND
              (GlitchDelay <= 0 ns) THEN
            NoGlitchDet := TRUE;
            Dly := Minimum( GlitchData.SchedTime-NOW, NewDelay );

        END IF;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        GlitchData.SchedTime := NOW+Dly;
        IF OldGlitch THEN
            OutSignal <= NewValue AFTER Dly;

        ELSIF NoGlitchDet THEN
            GlitchData.GlitchTime := NOW+Dly;
            OutSignal <= NewValue AFTER Dly;

        ELSE -- new glitch
            GlitchData.GlitchTime := GlitchMinTime ( GlitchData.GlitchTime,
                                                     NOW+GlitchDelay );

            IF (GlitchMode = MessagePlusX) OR
               (GlitchMode = MessageOnly) THEN
                ReportGlitch ( "VitalGlitchOnEvent", OutSignalName,
                               GlitchData.GlitchTime, GlitchData.SchedValue,
                               (Dly + NOW), NewValue,
                               MsgSeverity=>MsgSeverity );
            END IF;

            IF (GlitchMode = MessagePlusX) OR (GlitchMode = XOnly) THEN
                OutSignal <= 'X' AFTER GlitchData.GlitchTime-NOW;
                OutSignal <=  TRANSPORT NewValue AFTER Dly;
            ELSE
                OutSignal <= NewValue AFTER Dly;
            END IF;
        END IF;

        GlitchData.SchedValue := NewValue;
    END IF;

    RETURN;
END;

----------------------------------------------------------------------------
PROCEDURE VitalGlitchOnEvent (
        SIGNAL   OutSignal        : OUT   std_logic_vector;
        CONSTANT OutSignalName    : IN    STRING;
        VARIABLE GlitchData       : INOUT GlitchDataArrayType;
        CONSTANT NewValue         : IN    std_logic_vector;
        CONSTANT NewDelay         : IN    VitalTimeArray;
        CONSTANT GlitchMode       : IN    VitalGlitchModeType := MessagePlusX;
        CONSTANT GlitchDelay      : IN    VitalTimeArray;
        CONSTANT MsgSeverity      : IN    SEVERITY_LEVEL := WARNING
) IS

    ALIAS GlDataAlias  : GlitchDataArrayType(1 TO GlitchData'LENGTH)
                            IS GlitchData;
    ALIAS NewValAlias  : std_logic_vector(1 TO NewValue'LENGTH) IS NewValue;
    ALIAS GlDelayAlias : VitalTimeArray(1 TO GlitchDelay'LENGTH)
          IS GlitchDelay;
    ALIAS NewDelAlias  : VitalTimeArray(1 TO NewDelay'LENGTH) IS NewDelay;

    VARIABLE Index       : INTEGER := OutSignal'LEFT;
    VARIABLE Direction   : INTEGER;
    VARIABLE NoGlitchDet : BOOLEAN;
    VARIABLE OldGlitch   : BOOLEAN;
    VARIABLE Dly, GlDly  : TIME;

BEGIN
    IF (OutSignal'LEFT > OutSignal'RIGHT) THEN
        Direction := -1;
    ELSE
        Direction := 1;
    END IF;

    IF ( (OutSignal'LENGTH /=  GlitchData'LENGTH) OR
         (OutSignal'LENGTH /=    NewValue'LENGTH) OR
         (OutSignal'LENGTH /=    NewDelay'LENGTH) OR
         (OutSignal'LENGTH /= GlitchDelay'LENGTH) ) THEN
      VitalError ( "VitalGlitchOnEvent", ErrVctLng, OutSignalName );
      RETURN;
    END IF;

    -- a call to the scalar function cannot be made since the actual
    -- name associated with a signal parameter must be locally static
  FOR n IN 1 TO OutSignal'LENGTH LOOP

    NoGlitchDet := FALSE;
    OldGlitch   := FALSE;
    Dly := NewDelAlias(n);
```

Copyright © 2001 IEEE. All rights reserved.

```
      -- If nothing to schedule, just skip to next loop iteration
      IF NewDelAlias(n) < 0 ns THEN
          IF (NewValAlias(n) /=  GlDataAlias(n).SchedValue) THEN
              VitalError ( "VitalGlitchOnEvent", ErrNegDel, OutSignalName );
          END IF;
      ELSE
          -- If nothing currently scheduled (i.e. last scheduled
          -- transaction already occurred)
          IF GlDataAlias(n).SchedTime <= NOW THEN
              GlDataAlias(n).CurrentValue := GlDataAlias(n).SchedValue;
              IF (GlDelayAlias(n) <= 0 ns) THEN
                  -- Next iteration if no change in value
                  IF (NewValAlias(n) = GlDataAlias(n).SchedValue) THEN
                      Index := Index + Direction;
                      NEXT;
                  END IF;
                  -- since last transaction already occurred there is no glitch
                  NoGlitchDet := TRUE;
              END IF;

          -- Transaction currently scheduled - if glitch already happened
          ELSIF GlDataAlias(n).GlitchTime <= NOW THEN
              GlDataAlias(n).CurrentValue := 'X';
              OldGlitch := TRUE;
              IF (GlDataAlias(n).SchedValue = NewValAlias(n)) THEN
                  dly := Minimum( GlDataAlias(n).SchedTime-NOW,
                                  NewDelAlias(n) );
              END IF;

          -- Transaction currently scheduled
          ELSIF (GlDataAlias(n).SchedValue = NewValAlias(n)) AND
                (GlDataAlias(n).SchedTime = GlDataAlias(n).GlitchTime) AND
                (GlDelayAlias(n) <= 0 ns) THEN
                NoGlitchDet := TRUE;
                Dly := Minimum( GlDataAlias(n).SchedTime-NOW,
                                NewDelAlias(n) );
          END IF;

          -- update last scheduled transaction
          GlDataAlias(n).SchedTime := NOW+Dly;

          IF OldGlitch THEN
              OutSignal(Index) <= NewValAlias(n) AFTER Dly;
          ELSIF NoGlitchDet THEN
              -- if no glitch then update last glitch time
              -- and OutSignal(actual_index)
              GlDataAlias(n).GlitchTime := NOW+Dly;
              OutSignal(Index) <= NewValAlias(n) AFTER Dly;
          ELSE   -- new glitch
              GlDataAlias(n).GlitchTime := GlitchMinTime (
                                                GlDataAlias(n).GlitchTime,
                                                NOW+GlDelayAlias(n) );

              IF (GlitchMode = MessagePlusX) OR
                 (GlitchMode = MessageOnly) THEN
                  ReportGlitch ( "VitalGlitchOnEvent", OutSignalName,
                                 GlDataAlias(n).GlitchTime,
                                 GlDataAlias(n).SchedValue,
                                 (Dly + NOW), NewValAlias(n),
                                 Index, TRUE, MsgSeverity );
              END IF;

              IF (GlitchMode = MessagePlusX) OR (GlitchMode = XOnly) THEN
                  GlDly := GlDataAlias(n).GlitchTime - NOW;
                  OutSignal(Index) <= 'X' AFTER GlDly;
                  OutSignal(Index) <= TRANSPORT NewValAlias(n) AFTER Dly;
              ELSE
                  OutSignal(Index) <= NewValAlias(n) AFTER Dly;
              END IF;

          END IF; -- glitch / no-glitch
          GlDataAlias(n).SchedValue := NewValAlias(n);

      END IF; -- NewDelAlias(n) < 0 ns
      Index := Index + Direction;
    END LOOP;

      RETURN;
  END;
```

 Copyright © 2001 IEEE. All rights reserved.

```
---------------------------------------------------------------------------
-- ------------------------------------------------------------------------
--  PROCEDURE NAME  :  TruthOutputX01Z
--
--  PARAMETERS      :  table_out - output of table
--                     X01Zout   - output converted to X01Z
--                     err       - true if illegal character is encountered
--
--
--  DESCRIPTION     :  converts the output of a truth table to a valid
--                     std_ulogic
-- ------------------------------------------------------------------------
PROCEDURE TruthOutputX01Z (
        CONSTANT TableOut : IN VitalTruthSymbolType;
        VARIABLE X01Zout   : OUT std_ulogic;
        VARIABLE Err       : OUT BOOLEAN
) IS
    VARIABLE TempOut : std_ulogic;
BEGIN
    Err := FALSE;
    TempOut := TruthTableOutMap(TableOut);
    IF (TempOut = '-') THEN
        Err := TRUE;
        TempOut := 'X';
        VitalError ( "VitalTruthTable", ErrOutSym, To_TruthChar(TableOut));
    END IF;
    X01Zout := TempOut;
END;

-- ------------------------------------------------------------------------
--  PROCEDURE NAME  :  StateOutputX01Z
--
--  PARAMETERS      :  table_out - output of table
--                     prev_out  - previous output value
--                     X01Zout   - output cojnverted to X01Z
--                     err       - true if illegal character is encountered
--
--  DESCRIPTION     :  converts the output of a state table to a
--                     valid std_ulogic
-- ------------------------------------------------------------------------
PROCEDURE StateOutputX01Z (
        CONSTANT TableOut : IN VitalStateSymbolType;
        CONSTANT PrevOut  : IN std_ulogic;
        VARIABLE X01Zout   : OUT std_ulogic;
        VARIABLE Err       : OUT BOOLEAN
) IS
    VARIABLE TempOut : std_ulogic;
BEGIN
    Err := FALSE;
    TempOut := StateTableOutMap(TableOut);
    IF (TempOut = '-') THEN
        Err := TRUE;
        TempOut := 'X';
        VitalError ( "VitalStateTable", ErrOutSym, To_StateChar(TableOut));
    ELSIF (TempOut = 'W') THEN
        TempOut := To_X01Z(PrevOut);
    END IF;
    X01Zout := TempOut;
END;

-- ------------------------------------------------------------------------
-- PROCEDURE NAME:  StateMatch
--
-- PARAMETERS    :  symbol       - symbol from state table
--                  in2          - input from VitalStateTble procedure
--                                 to state table
--                  in2LastValue - previous value of input
--                  state        - false if the symbol is from the input
--                                  portion of the table,
--                                 true if the symbol is from the state
--                                  portion of the table
--                  Err          - true if symbol is not a valid input symbol
--                  ReturnValue  - true if match occurred
--
-- DESCRIPTION   :  This procedure sets ReturnValue to true if in2 matches
--                  symbol (from the state table).  If symbol is an edge
--                  value edge is set to true and in2 and in2LastValue are
--                  checked against symbol.  Err is set to true if symbol
--                  is an invalid value for the input portion of the state
--                  table.
--
```

Copyright © 2001 IEEE. All rights reserved.

```
-- ------------------------------------------------------------------------
PROCEDURE StateMatch (
        CONSTANT Symbol       : IN VitalStateSymbolType;
        CONSTANT in2          : IN std_ulogic;
        CONSTANT in2LastValue : IN std_ulogic;
        CONSTANT State        : IN BOOLEAN;
        VARIABLE Err          : OUT BOOLEAN;
        VARIABLE ReturnValue  : OUT BOOLEAN
) IS
BEGIN
    IF (State) THEN
        IF (NOT ValidStateTableState(Symbol)) THEN
            VitalError ( "VitalStateTable", ErrStaSym, To_StateChar(Symbol));
            Err := TRUE;
            ReturnValue := FALSE;
        ELSE
            Err := FALSE;
            ReturnValue := StateTableMatch(in2LastValue, in2, Symbol);
        END IF;
    ELSE
        IF (NOT ValidStateTableInput(Symbol) ) THEN
            VitalError ( "VitalStateTable", ErrInpSym, To_StateChar(Symbol));
            Err := TRUE;
            ReturnValue := FALSE;
        ELSE
            ReturnValue := StateTableMatch(in2LastValue, in2, Symbol);
            Err := FALSE;
        END IF;
    END IF;
END;

-- ------------------------------------------------------------------------
-- FUNCTION NAME:  StateTableLookUp
--
-- PARAMETERS   :  StateTable     - state table
--                 PresentDataIn  - current inputs
--                 PreviousDataIn - previous inputs and states
--                 NumStates      - number of state variables
--                 PresentOutputs - current state and current outputs
--
-- DESCRIPTION  :  This function is used to find the output of the
--                 StateTable corresponding to a given set of inputs.
--
-- ------------------------------------------------------------------------
FUNCTION StateTableLookUp (
        CONSTANT StateTable     : VitalStateTableType;
        CONSTANT PresentDataIn  : std_logic_vector;
        CONSTANT PreviousDataIn : std_logic_vector;
        CONSTANT NumStates      : NATURAL;
        CONSTANT PresentOutputs : std_logic_vector
      ) RETURN std_logic_vector IS

    CONSTANT InputSize    : INTEGER := PresentDataIn'LENGTH;
    CONSTANT NumInputs    : INTEGER := InputSize + NumStates - 1;
    CONSTANT TableEntries : INTEGER := StateTable'LENGTH(1);
    CONSTANT TableWidth   : INTEGER := StateTable'LENGTH(2);
    CONSTANT OutSize      : INTEGER := TableWidth - InputSize - NumStates;
    VARIABLE Inputs       : std_logic_vector(0 TO NumInputs);
    VARIABLE PrevInputs   : std_logic_vector(0 TO NumInputs)
                            := (OTHERS => 'X');
    VARIABLE ReturnValue  : std_logic_vector(0 TO (OutSize-1))
                            := (OTHERS => 'X');
    VARIABLE Temp   : std_ulogic;
    VARIABLE Match  : BOOLEAN;
    VARIABLE Err    : BOOLEAN := FALSE;

    -- This needs to be done since the TableLookup arrays must be
    -- ascending starting with 0
    VARIABLE TableAlias   : VitalStateTableType(0 TO TableEntries - 1,
                                                0 TO TableWidth - 1)
                            := StateTable;

BEGIN
    Inputs(0 TO InputSize-1) := PresentDataIn;
    Inputs(InputSize TO NumInputs) := PresentOutputs(0 TO NumStates - 1);
    PrevInputs(0 TO InputSize - 1) := PreviousDataIn(0 TO InputSize - 1);

  ColLoop: -- Compare each entry in the table
    FOR i IN TableAlias'RANGE(1) LOOP

    RowLoop: -- Check each element of the entry
```

 Copyright © 2001 IEEE. All rights reserved.

```
      FOR j IN 0 TO InputSize + NumStates  LOOP

        IF (j = InputSize + NumStates) THEN        -- a match occurred
            FOR k IN 0 TO Minimum(OutSize, PresentOutputs'LENGTH)-1  LOOP
                StateOutputX01Z (
                        TableAlias(i, TableWidth - k - 1),
                        PresentOutputs(PresentOutputs'LENGTH - k - 1),
                        Temp, Err);
                ReturnValue(OutSize - k - 1) := Temp;
                IF (Err) THEN
                    ReturnValue := (OTHERS => 'X');
                    RETURN ReturnValue;
                END IF;
            END LOOP;
            RETURN ReturnValue;
        END IF;

        StateMatch ( TableAlias(i,j),
                     Inputs(j), PrevInputs(j),
                     j >= InputSize, Err, Match);
        EXIT RowLoop WHEN NOT(Match);
        EXIT ColLoop WHEN Err;
      END LOOP RowLoop;
    END LOOP ColLoop;

    ReturnValue := (OTHERS => 'X');
    RETURN ReturnValue;
END;

--------------------------------------------------------------------
-- to_ux01z
-------------------------------------------------------------------
FUNCTION To_UX01Z  ( s : std_ulogic
      ) RETURN  UX01Z IS
BEGIN
    RETURN cvt_to_ux01z (s);
END;

---------------------------------------------------------------------------
-- Function  : GetEdge
-- Purpose   : Converts transitions on a given input signal into a
--             enumeration value representing the transition or level
--             of the signal.
--
--    previous "value"   current "value"     :=   "edge"
--   ---------------------------------------------------------
--     '1' | 'H'          '1' | 'H'                 '1'    level, no edge
--     '0' | 'L'          '1' | 'H'                 '/'    rising edge
--      others            '1' | 'H'                 'R'    rising from X
--
--     '1' | 'H'          '0' | 'L'                 '\'    falling egde
--     '0' | 'L'          '0' | 'L'                 '0'    level, no edge
--      others            '0' | 'L'                 'F'    falling from X
--
--     'X' | 'W' | '-'    'X' | 'W' | '-'           'X'    unknown (X) level
--     'Z'                'Z'                       'X'    unknown (X) level
--     'U'                'U'                       'U'    'U' level
--
--     '1' | 'H'           others                   'f'    falling to X
--     '0' | 'L'           others                   'r'    rising to X
--     'X' | 'W' | '-'    'U' | 'Z'                 'x'    unknown (X) edge
--     'Z'                'X' | 'W' | '-' | 'U'     'x'    unknown (X) edge
--     'U'                'X' | 'W' | '-' | 'Z'     'x'    unknown (X) edge
--
---------------------------------------------------------------------------
FUNCTION GetEdge (
        SIGNAL      s : IN    std_logic
      ) RETURN EdgeType IS
BEGIN
    IF (s'EVENT)
        THEN RETURN LogicToEdge  ( s'LAST_VALUE, s );
        ELSE RETURN LogicToLevel ( s );
    END IF;
END;

---------------------------------------------------------------------------
PROCEDURE GetEdge (
        SIGNAL       s : IN    std_logic_vector;
        VARIABLE LastS : INOUT std_logic_vector;
        VARIABLE  Edge :   OUT EdgeArray ) IS
```

Copyright © 2001 IEEE. All rights reserved.

```
    ALIAS     sAlias : std_logic_vector ( 1 TO     s'LENGTH ) IS s;
    ALIAS LastSAlias : std_logic_vector ( 1 TO LastS'LENGTH ) IS LastS;
    ALIAS  EdgeAlias : EdgeArray ( 1 TO  Edge'LENGTH ) IS Edge;
BEGIN
    IF s'LENGTH /= LastS'LENGTH OR
       s'LENGTH /=  Edge'LENGTH THEN
        VitalError ( "GetEdge", ErrVctLng, "s, LastS, Edge" );
    END IF;

    FOR n IN 1 TO s'LENGTH LOOP
        EdgeAlias(n)  := LogicToEdge( LastSAlias(n), sAlias(n) );
        LastSAlias(n) := sAlias(n);
    END LOOP;
END;

---------------------------------------------------------------------------
FUNCTION  ToEdge     ( Value         : IN std_logic
      ) RETURN EdgeType IS
BEGIN
    RETURN LogicToLevel( Value );
END;

-- Note: This function will likely be replaced by S'DRIVING_VALUE in VHDL'92
----------------------------------------------------------------------------
IMPURE FUNCTION CurValue (
        CONSTANT GlitchData : IN  GlitchDataType
      ) RETURN std_logic IS
BEGIN
    IF NOW >= GlitchData.SchedTime THEN
        RETURN GlitchData.SchedValue;
    ELSIF NOW >= GlitchData.GlitchTime THEN
        RETURN 'X';
    ELSE
        RETURN GlitchData.CurrentValue;
    END IF;
END;
---------------------------------------------------------------------------
IMPURE FUNCTION CurValue (
        CONSTANT GlitchData : IN  GlitchDataArrayType
      ) RETURN std_logic_vector IS
    VARIABLE Result : std_logic_vector(GlitchData'RANGE);
BEGIN
    FOR n IN GlitchData'RANGE LOOP
        IF NOW >= GlitchData(n).SchedTime THEN
            Result(n) := GlitchData(n).SchedValue;
        ELSIF NOW >= GlitchData(n).GlitchTime THEN
            Result(n) := 'X';
        ELSE
            Result(n) := GlitchData(n).CurrentValue;
        END IF;
    END LOOP;
    RETURN Result;
END;

---------------------------------------------------------------------------
-- function calculation utilities
---------------------------------------------------------------------------

---------------------------------------------------------------------------
-- Function   : VitalSame
-- Returns    : VitalSame compares the state (UX01) of two logic value. A
--              value of 'X' is returned if the values are different.  The
--              common value is returned if the values are equal.
-- Purpose    : When the result of a logic model may be either of two
--              separate input values (eg. when the select on a MUX is 'X'),
--              VitalSame may be used to determine if the result needs to
--              be 'X'.
-- Arguments  : See the declarations below...
---------------------------------------------------------------------------
FUNCTION VitalSame (
        CONSTANT a, b : IN std_ulogic
      ) RETURN std_ulogic IS
BEGIN
    IF To_UX01(a) = To_UX01(b)
        THEN RETURN To_UX01(a);
        ELSE RETURN 'X';
    END IF;
END;

---------------------------------------------------------------------------
-- delay selection utilities
```

 Copyright © 2001 IEEE. All rights reserved.

```
---------------------------------------------------------------------------

---------------------------------------------------------------------------
-- Procedure  : BufPath, InvPath
--
-- Purpose    : BufPath and InvPath compute output change times, based on
--              a change on an input port. The computed output change times
--              returned in the composite parameter 'schd'.
--
--              BufPath and InpPath are used together with the delay path
--              selection functions (GetSchedDelay, VitalAND, VitalOR... )
--              The 'schd' value from each of the input ports of a model are
--              combined by the delay selection functions (VitalAND,
--              VitalOR, ...). The GetSchedDelay procedure converts the
--              combined output changes times to the single delay (delta
--              time) value for scheduling the output change (passed to
--              VitalGlitchOnEvent).
--
--              The values in 'schd' are: (absolute times)
--                inp0  :  time of output change due to input change to 0
--                inp1  :  time of output change due to input change to 1
--                inpX  :  time of output change due to input change to X
--                glch0 :  time of output glitch due to input change to 0
--                glch1 :  time of output glitch due to input change to 1
--
--              The output times are computed from the model INPUT value
--              and not the final value.  For this reason, 'BufPath' should
--              be used to compute the output times for a non-inverting
--              delay paths and 'InvPath' should be used to compute the
--              ouput times for inverting delay paths. Delay paths which
--              include both non-inverting and paths require usage of both
--              'BufPath' and 'InvPath'. (IE this is needed for the
--              select->output path of a MUX -- See the VitalMUX model).
--
--
-- Parameters : schd....... Computed output result times. (INOUT parameter
--                          modified only on input edges)
--              Iedg....... Input port edge/level value.
--               tpd....... Propagation delays from this input
--
---------------------------------------------------------------------------

PROCEDURE BufPath (
        VARIABLE Schd : INOUT SchedType;
        CONSTANT Iedg : IN    EdgeType;
        CONSTANT  tpd : IN    VitalDelayType01
) IS
BEGIN
  CASE Iedg IS
    WHEN '0'|'1'  => NULL;                   -- no edge: no timing update
    WHEN '/'|'R'  => Schd.inp0 := TIME'HIGH;
                     Schd.inp1 := NOW + tpd(tr01);  Schd.Glch1 := Schd.inp1;
                     Schd.InpX := Schd.inp1;
    WHEN '\'|'F'  => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := NOW + tpd(tr10);  Schd.Glch0 := Schd.inp0;
                     Schd.InpX := Schd.inp0;
    WHEN 'r'      => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := TIME'HIGH;
                     Schd.InpX := NOW + tpd(tr01);
    WHEN 'f'      => Schd.inp0 := TIME'HIGH;
                     Schd.inp1 := TIME'HIGH;
                     Schd.InpX := NOW + tpd(tr10);
    WHEN 'x'      => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := TIME'HIGH;
                     -- update for X->X change
                     Schd.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
    WHEN OTHERS   => NULL;                   -- no timing change
  END CASE;
END;

PROCEDURE BufPath (
        VARIABLE Schd : INOUT SchedArray;
        CONSTANT Iedg : IN    EdgeArray;
        CONSTANT  tpd : IN    VitalDelayArrayType01
) IS
BEGIN
  FOR n IN Schd'RANGE LOOP
    CASE Iedg(n) IS
      WHEN '0'|'1'  => NULL;                   -- no edge: no timing update
      WHEN '/'|'R'  => Schd(n).inp0 := TIME'HIGH;
                       Schd(n).inp1 := NOW + tpd(n)(tr01);
```

Copyright © 2001 IEEE. All rights reserved.

```
                        Schd(n).Glch1 := Schd(n).inp1;
                        Schd(n).InpX := Schd(n).inp1;
        WHEN '\' |' F'  => Schd(n).inp1 := TIME' HIGH;
                        Schd(n).inp0 := NOW + tpd(n)(tr10);
                        Schd(n).Glch0 := Schd(n).inp0;
                        Schd(n).InpX := Schd(n).inp0;
        WHEN 'r'      => Schd(n).inp1 := TIME' HIGH;
                        Schd(n).inp0 := TIME' HIGH;
                        Schd(n).InpX := NOW + tpd(n)(tr01);
        WHEN 'f'      => Schd(n).inp0 := TIME' HIGH;
                        Schd(n).inp1 := TIME' HIGH;
                        Schd(n).InpX := NOW + tpd(n)(tr10);
        WHEN 'x'      => Schd(n).inp1 := TIME' HIGH;
                        Schd(n).inp0 := TIME' HIGH;
                        -- update for X->X change
                        Schd(n).InpX := NOW + Minimum ( tpd(n)(tr10),
                                                        tpd(n)(tr01) );
        WHEN OTHERS  => NULL;                   -- no timing change
      END CASE;
    END LOOP;
  END;

  PROCEDURE InvPath (
          VARIABLE Schd : INOUT SchedType;
          CONSTANT Iedg : IN    EdgeType;
          CONSTANT  tpd : IN    VitalDelayType01
  ) IS
  BEGIN
    CASE Iedg IS
      WHEN '0' |' 1'  => NULL;                   -- no edge: no timing update
      WHEN '/' |' R'  => Schd.inp0 := TIME' HIGH;
                      Schd.inp1 := NOW + tpd(tr10);  Schd.Glch1 := Schd.inp1;
                      Schd.InpX := Schd.inp1;
      WHEN '\' |' F'  => Schd.inp1 := TIME' HIGH;
                      Schd.inp0 := NOW + tpd(tr01);  Schd.Glch0 := Schd.inp0;
                      Schd.InpX := Schd.inp0;
      WHEN 'r'      => Schd.inp1 := TIME' HIGH;
                      Schd.inp0 := TIME' HIGH;
                      Schd.InpX := NOW + tpd(tr10);
      WHEN 'f'      => Schd.inp0 := TIME' HIGH;
                      Schd.inp1 := TIME' HIGH;
                      Schd.InpX := NOW + tpd(tr01);
      WHEN 'x'      => Schd.inp1 := TIME' HIGH;
                      Schd.inp0 := TIME' HIGH;
                      -- update for X->X change
                      Schd.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
      WHEN OTHERS  => NULL;                   -- no timing change
    END CASE;
  END;

  PROCEDURE InvPath (
          VARIABLE Schd : INOUT SchedArray;
          CONSTANT Iedg : IN    EdgeArray;
          CONSTANT  tpd : IN    VitalDelayArrayType01
  ) IS
  BEGIN
    FOR n IN Schd' RANGE LOOP
      CASE Iedg(n) IS
        WHEN '0' |' 1'  => NULL;                   -- no edge: no timing update
        WHEN '/' |' R'  => Schd(n).inp0 := TIME' HIGH;
                        Schd(n).inp1 := NOW + tpd(n)(tr10);
                        Schd(n).Glch1 := Schd(n).inp1;
                        Schd(n).InpX := Schd(n).inp1;
        WHEN '\' |' F'  => Schd(n).inp1 := TIME' HIGH;
                        Schd(n).inp0 := NOW + tpd(n)(tr01);
                        Schd(n).Glch0 := Schd(n).inp0;
                        Schd(n).InpX := Schd(n).inp0;
        WHEN 'r'      => Schd(n).inp1 := TIME' HIGH;
                        Schd(n).inp0 := TIME' HIGH;
                        Schd(n).InpX := NOW + tpd(n)(tr10);
        WHEN 'f'      => Schd(n).inp0 := TIME' HIGH;
                        Schd(n).inp1 := TIME' HIGH;
                        Schd(n).InpX := NOW + tpd(n)(tr01);
        WHEN 'x'      => Schd(n).inp1 := TIME' HIGH;
                        Schd(n).inp0 := TIME' HIGH;
                        -- update for X->X change
                        Schd(n).InpX := NOW + Minimum ( tpd(n)(tr10),
                                                        tpd(n)(tr01) );
        WHEN OTHERS  => NULL;                   -- no timing change
      END CASE;
    END LOOP;
```

 Copyright © 2001 IEEE. All rights reserved.

```
END;

---------------------------------------------------------------------------
-- Procedure  : BufEnab, InvEnab
--
-- Purpose    : BufEnab and InvEnab compute output change times, from a
--              change on an input enable port for a 3-state driver. The
--              computed output change times are returned in the composite
--              parameters 'schd1', 'schd0'.
--
--              BufEnab and InpEnab are used together with the delay path
--              selection functions (GetSchedDelay, VitalAND, VitalOR... )
--              The 'schd' value from each of the non-enable input ports of
--              a model (See BufPath, InvPath) are combined using the delay
--              selection functions (VitalAND,  VitalOR, ...). The
--              GetSchedDelay procedure combines the output times on the
--              enable path with the output times from the data path(s) and
--              computes the single delay (delta time) value for scheduling
--              the output change (passed to VitalGlitchOnEvent)
--
--              The values in 'schd*' are: (absolute times)
--                inp0  :  time of output change due to input change to 0
--                inp1  :  time of output change due to input change to 1
--                inpX  :  time of output change due to input change to X
--                glch0 :  time of output glitch due to input change to 0
--                glch1 :  time of output glitch due to input change to 1
--
--              'schd1' contains output times for 1->Z, Z->1 transitions.
--              'schd0' contains output times for 0->Z, Z->0 transitions.
--
--              'BufEnab' is used for computing the output times for an
--              high asserted enable (output 'Z' for enable='0').
--              'InvEnab' is used for computing the output times for an
--              low asserted enable (output 'Z' for enable='1').
--
--              Note: separate 'schd1', 'schd0' parameters are generated
--                    so that the combination of the delay paths from
--                    multiple enable signals may be combined using the
--                    same functions/operators used in combining separate
--                    data paths. (See exampe 2 below)
--
--
-- Parameters : schd1...... Computed output result times for 1->Z, Z->1
--                          transitions. This parameter is modified only on
--                          input edge values (events).
--              schd0...... Computed output result times for 0->Z, 0->1
--                          transitions. This parameter is modified only on
--                          input edge values (events).
--              Iedg....... Input port edge/level value.
--               tpd....... Propagation delays for the enable -> output path.
--
---------------------------------------------------------------------------
PROCEDURE BufEnab (
        VARIABLE Schd1 : INOUT SchedType;
        VARIABLE Schd0 : INOUT SchedType;
        CONSTANT  Iedg : IN    EdgeType;
        CONSTANT   tpd : IN    VitalDelayType01Z
) IS
BEGIN
  CASE Iedg IS
    WHEN '0'|'1'  => NULL;                    -- no edge: no timing update
    WHEN '/'|'R'  => Schd1.inp0 := TIME'HIGH;
                     Schd1.inp1 := NOW + tpd(trz1);
                     Schd1.Glch1 := Schd1.inp1;
                     Schd1.InpX := Schd1.inp1;
                     Schd0.inp0 := TIME'HIGH;
                     Schd0.inp1 := NOW + tpd(trz0);
                     Schd0.Glch1 := Schd0.inp1;
                     Schd0.InpX := Schd0.inp1;
    WHEN '\'|'F'  => Schd1.inp1 := TIME'HIGH;
                     Schd1.inp0 := NOW + tpd(tr1z);
                     Schd1.Glch0 := Schd1.inp0;
                     Schd1.InpX := Schd1.inp0;
                     Schd0.inp1 := TIME'HIGH;
                     Schd0.inp0 := NOW + tpd(tr0z);
                     Schd0.Glch0 := Schd0.inp0;
                     Schd0.InpX := Schd0.inp0;
    WHEN 'r'      => Schd1.inp1 := TIME'HIGH;
                     Schd1.inp0 := TIME'HIGH;
                     Schd1.InpX := NOW + tpd(trz1);
                     Schd0.inp1 := TIME'HIGH;
```

Copyright © 2001 IEEE. All rights reserved.

```
                      Schd0.inp0 := TIME'HIGH;
                      Schd0.InpX := NOW + tpd(trz0);
    WHEN 'f'      => Schd1.inp0 := TIME'HIGH;
                      Schd1.inp1 := TIME'HIGH;
                      Schd1.InpX := NOW + tpd(tr1z);
                      Schd0.inp0 := TIME'HIGH;
                      Schd0.inp1 := TIME'HIGH;
                      Schd0.InpX := NOW + tpd(tr0z);
    WHEN 'x'      => Schd1.inp0 := TIME'HIGH;
                      Schd1.inp1 := TIME'HIGH;
                      Schd1.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
                      Schd0.inp0 := TIME'HIGH;
                      Schd0.inp1 := TIME'HIGH;
                      Schd0.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
    WHEN OTHERS  => NULL;                    -- no timing change
  END CASE;
END;

PROCEDURE InvEnab (
        VARIABLE Schd1 : INOUT SchedType;
        VARIABLE Schd0 : INOUT SchedType;
        CONSTANT  Iedg : IN    EdgeType;
        CONSTANT   tpd : IN    VitalDelayType01Z
) IS
BEGIN
  CASE Iedg IS
    WHEN '0'|'1'  => NULL;                    -- no edge: no timing update
    WHEN '/'|'R'  => Schd1.inp0 := TIME'HIGH;
                      Schd1.inp1 := NOW + tpd(tr1z);
                      Schd1.Glch1 := Schd1.inp1;
                      Schd1.InpX := Schd1.inp1;
                      Schd0.inp0 := TIME'HIGH;
                      Schd0.inp1 := NOW + tpd(tr0z);
                      Schd0.Glch1 := Schd0.inp1;
                      Schd0.InpX := Schd0.inp1;
    WHEN '\'|'F'  => Schd1.inp1 := TIME'HIGH;
                      Schd1.inp0 := NOW + tpd(trz1);
                      Schd1.Glch0 := Schd1.inp0;
                      Schd1.InpX := Schd1.inp0;
                      Schd0.inp1 := TIME'HIGH;
                      Schd0.inp0 := NOW + tpd(trz0);
                      Schd0.Glch0 := Schd0.inp0;
                      Schd0.InpX := Schd0.inp0;
    WHEN 'r'      => Schd1.inp1 := TIME'HIGH;
                      Schd1.inp0 := TIME'HIGH;
                      Schd1.InpX := NOW + tpd(tr1z);
                      Schd0.inp1 := TIME'HIGH;
                      Schd0.inp0 := TIME'HIGH;
                      Schd0.InpX := NOW + tpd(tr0z);
    WHEN 'f'      => Schd1.inp0 := TIME'HIGH;
                      Schd1.inp1 := TIME'HIGH;
                      Schd1.InpX := NOW + tpd(trz1);
                      Schd0.inp0 := TIME'HIGH;
                      Schd0.inp1 := TIME'HIGH;
                      Schd0.InpX := NOW + tpd(trz0);
    WHEN 'x'      => Schd1.inp0 := TIME'HIGH;
                      Schd1.inp1 := TIME'HIGH;
                      Schd1.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
                      Schd0.inp0 := TIME'HIGH;
                      Schd0.inp1 := TIME'HIGH;
                      Schd0.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
    WHEN OTHERS  => NULL;                    -- no timing change
  END CASE;
END;

---------------------------------------------------------------------------
-- Procedure  : GetSchedDelay
--
-- Purpose    : GetSchedDelay computes the final delay (incremental) for
--              for scheduling an output signal.  The delay is computed
--              from the absolute output times in the 'NewSched' parameter.
--              (See BufPath, InvPath).
--
--              Computation of the output delay for non-3_state outputs
--              consists of selection the appropriate output time based
--              on the new output value 'NewValue' and subtracting 'NOW'
--              to convert to an incremental delay value.
--
--              The Computation of the output delay for 3_state output
--              also includes combination of the enable path delay with
--              the date path delay.
```

 Copyright © 2001 IEEE. All rights reserved.

```
--
-- Parameters : NewDelay... Returned output delay value.
--              GlchDelay.. Returned output delay for the start of a glitch.
--              NewValue... New output value.
--              CurValue... Current value of the output.
--              NewSched... Composite containing the combined absolute
--                          output times from the data inputs.
--              EnSched1... Composite containing the combined absolute
--                          output times from the enable input(s).
--                          (for a 3_state output transitions 1->Z, Z->1)
--              EnSched0... Composite containing the combined absolute
--                          output times from the enable input(s).
--                          (for a 3_state output transitions 0->Z, Z->0)
--
---------------------------------------------------------------------------
PROCEDURE GetSchedDelay (
        VARIABLE   NewDelay : OUT TIME;
        VARIABLE  GlchDelay : OUT TIME;
        CONSTANT   NewValue : IN  std_ulogic;
        CONSTANT   CurValue : IN  std_ulogic;
        CONSTANT   NewSched : IN  SchedType
) IS
    VARIABLE Tim, Glch : TIME;
BEGIN

    CASE To_UX01(NewValue) IS
      WHEN '0'     => Tim  := NewSched.inp0;
                      Glch := NewSched.Glch1;
      WHEN '1'     => Tim  := NewSched.inp1;
                      Glch := NewSched.Glch0;
      WHEN OTHERS => Tim  := NewSched.InpX;
                      Glch := -1 ns;
    END CASE;
    IF (CurValue /= NewValue)
      THEN Glch := -1 ns;
    END IF;

    NewDelay  := Tim  - NOW;
    IF Glch < 0 ns
        THEN GlchDelay := Glch;
        ELSE GlchDelay := Glch - NOW;
    END IF; -- glch < 0 ns
END;

PROCEDURE GetSchedDelay (
        VARIABLE   NewDelay : OUT VitalTimeArray;
        VARIABLE  GlchDelay : OUT VitalTimeArray;
        CONSTANT   NewValue : IN  std_logic_vector;
        CONSTANT   CurValue : IN  std_logic_vector;
        CONSTANT   NewSched : IN  SchedArray
) IS
    VARIABLE Tim, Glch : TIME;
    ALIAS  NewDelayAlias : VitalTimeArray( NewDelay'LENGTH DOWNTO 1)
           IS NewDelay;
    ALIAS GlchDelayAlias : VitalTimeArray(GlchDelay'LENGTH DOWNTO 1)
           IS GlchDelay;
    ALIAS  NewSchedAlias : SchedArray( NewSched'LENGTH DOWNTO 1)
           IS NewSched;
    ALIAS  NewValueAlias : std_logic_vector (  NewValue'LENGTH DOWNTO 1 )
                            IS  NewValue;
    ALIAS  CurValueAlias : std_logic_vector (  CurValue'LENGTH DOWNTO 1 )
                            IS  CurValue;
BEGIN
  FOR n IN NewDelay'LENGTH DOWNTO 1 LOOP
    CASE To_UX01(NewValueAlias(n)) IS
      WHEN '0'     => Tim  := NewSchedAlias(n).inp0;
                      Glch := NewSchedAlias(n).Glch1;
      WHEN '1'     => Tim  := NewSchedAlias(n).inp1;
                      Glch := NewSchedAlias(n).Glch0;
      WHEN OTHERS => Tim  := NewSchedAlias(n).InpX;
                      Glch := -1 ns;
    END CASE;
    IF (CurValueAlias(n) /= NewValueAlias(n))
      THEN Glch := -1 ns;
    END IF;

    NewDelayAlias(n) := Tim  - NOW;
    IF Glch < 0 ns
        THEN GlchDelayAlias(n) := Glch;
        ELSE GlchDelayAlias(n) := Glch - NOW;
    END IF; -- glch < 0 ns
```

Copyright © 2001 IEEE. All rights reserved.

```
  END LOOP;
  RETURN;
END;

PROCEDURE GetSchedDelay (
        VARIABLE   NewDelay : OUT TIME;
        VARIABLE  GlchDelay : OUT TIME;
        CONSTANT   NewValue : IN  std_ulogic;
        CONSTANT   CurValue : IN  std_ulogic;
        CONSTANT   NewSched : IN  SchedType;
        CONSTANT   EnSched1 : IN  SchedType;
        CONSTANT   EnSched0 : IN  SchedType
) IS
    SUBTYPE v2 IS std_logic_vector(0 TO 1);
    VARIABLE Tim, Glch : TIME;
BEGIN

    CASE v2'(To_X01Z(CurValue) & To_X01Z(NewValue)) IS
      WHEN "00"    => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                      Glch := GlitchMinTime(NewSched.Glch1,EnSched0.Glch0);
      WHEN "01"    => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                      Glch := EnSched1.Glch0;
      WHEN "0Z"    => Tim  := EnSched0.inp0;
                      Glch := NewSched.Glch1;
      WHEN "0X"    => Tim  := Maximum (NewSched.InpX, EnSched1.InpX);
                      Glch := 0 ns;
      WHEN "10"    => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                      Glch := EnSched0.Glch0;
      WHEN "11"    => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                      Glch := GlitchMinTime(NewSched.Glch0,EnSched1.Glch0);
      WHEN "1Z"    => Tim  := EnSched1.inp0;
                      Glch := NewSched.Glch0;
      WHEN "1X"    => Tim  := Maximum (NewSched.InpX, EnSched0.InpX);
                      Glch := 0 ns;
      WHEN "Z0"    => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                      IF NewSched.Glch0 > NOW
                        THEN Glch := Maximum(NewSched.Glch1,EnSched1.inp1);
                        ELSE Glch := 0 ns;
                      END IF;
      WHEN "Z1"    => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                      IF NewSched.Glch1 > NOW
                        THEN Glch := Maximum(NewSched.Glch0,EnSched0.inp1);
                        ELSE Glch := 0 ns;
                      END IF;
      WHEN "ZX"    => Tim  := Maximum (NewSched.InpX, EnSched1.InpX);
                      Glch := 0 ns;
      WHEN "ZZ"    => Tim  := Maximum (EnSched1.InpX, EnSched0.InpX);
                      Glch := 0 ns;
      WHEN "X0"    => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                      Glch := 0 ns;
      WHEN "X1"    => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                      Glch := 0 ns;
      WHEN "XZ"    => Tim  := Maximum (EnSched1.InpX, EnSched0.InpX);
                      Glch := 0 ns;
      WHEN OTHERS  => Tim  := Maximum (NewSched.InpX, EnSched1.InpX);
                      Glch := 0 ns;

    END CASE;
    NewDelay  := Tim  - NOW;
    IF Glch < 0 ns
        THEN GlchDelay := Glch;
        ELSE GlchDelay := Glch - NOW;
    END IF; -- glch < 0 ns
END;

---------------------------------------------------------------------------
-- Operators and Functions for combination (selection) of path delays
-- > These functions support selection of the "appripriate" path delay
--   dependent on the logic function.
-- > These functions only "select" from the possable output times. No
--   calculation (addition) of delays is performed.
-- > See description of 'BufPath', 'InvPath' and 'GetSchedDelay'
-- > See primitive PROCEDURE models for examples.
---------------------------------------------------------------------------

FUNCTION "not"  (
        CONSTANT a : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    z.inp1  := a.inp0 ;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        z.inp0  := a.inp1 ;
        z.InpX  := a.InpX ;
        z.Glch1 := a.Glch0;
        z.Glch0 := a.Glch1;
        RETURN (z);
    END;

    IMPURE FUNCTION "and"  (
            CONSTANT a, b : IN SchedType
          ) RETURN SchedType IS
        VARIABLE z : SchedType;
    BEGIN
        z.inp1  := Maximum   ( a.inp1 , b.inp1  );
        z.inp0  := Minimum   ( a.inp0 , b.inp0  );
        z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
        z.Glch1 := Maximum   ( a.Glch1, b.Glch1 );
        z.Glch0 := GlitchMinTime ( a.Glch0, b.Glch0 );
        RETURN (z);
    END;

    IMPURE FUNCTION "or"   (
            CONSTANT a, b : IN SchedType
          ) RETURN SchedType IS
        VARIABLE z : SchedType;
    BEGIN
        z.inp0  := Maximum   ( a.inp0 , b.inp0  );
        z.inp1  := Minimum   ( a.inp1 , b.inp1  );
        z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
        z.Glch0 := Maximum   ( a.Glch0, b.Glch0 );
        z.Glch1 := GlitchMinTime ( a.Glch1, b.Glch1 );
        RETURN (z);
    END;

    IMPURE FUNCTION "nand" (
            CONSTANT a, b : IN SchedType
          ) RETURN SchedType IS
        VARIABLE z : SchedType;
    BEGIN
        z.inp0  := Maximum   ( a.inp1 , b.inp1  );
        z.inp1  := Minimum   ( a.inp0 , b.inp0  );
        z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
        z.Glch0 := Maximum   ( a.Glch1, b.Glch1 );
        z.Glch1 := GlitchMinTime ( a.Glch0, b.Glch0 );
        RETURN (z);
    END;

    IMPURE FUNCTION "nor"  (
            CONSTANT a, b : IN SchedType
          ) RETURN SchedType IS
        VARIABLE z : SchedType;
    BEGIN
        z.inp1  := Maximum   ( a.inp0 , b.inp0  );
        z.inp0  := Minimum   ( a.inp1 , b.inp1  );
        z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
        z.Glch1 := Maximum   ( a.Glch0, b.Glch0 );
        z.Glch0 := GlitchMinTime ( a.Glch1, b.Glch1 );
        RETURN (z);
    END;

    -- ---------------------------------------------------------------------------
    -- Delay Calculation for 2-bit Logical gates.
    -- ---------------------------------------------------------------------------
    IMPURE FUNCTION VitalXOR2   (
            CONSTANT ab,ai, bb,bi : IN SchedType
          ) RETURN SchedType IS
        VARIABLE z : SchedType;
    BEGIN
        -- z = (a AND b) NOR (a NOR b)
        z.inp1  :=   Maximum (  Minimum (ai.inp0 , bi.inp0 ),
                                Minimum (ab.inp1 , bb.inp1 ) );
        z.inp0  :=   Minimum (  Maximum (ai.inp1 , bi.inp1 ),
                                Maximum (ab.inp0 , bb.inp0 ) );
        z.InpX  :=   Maximum (  Maximum (ai.InpX , bi.InpX ),
                                Maximum (ab.InpX , bb.InpX ) );
        z.Glch1 :=   Maximum (GlitchMinTime (ai.Glch0, bi.Glch0),
                              GlitchMinTime (ab.Glch1, bb.Glch1) );
        z.Glch0 := GlitchMinTime (  Maximum (ai.Glch1, bi.Glch1),
                                Maximum (ab.Glch0, bb.Glch0) );
        RETURN (z);
    END;
```

Copyright © 2001 IEEE. All rights reserved.

```
    IMPURE FUNCTION VitalXNOR2  (
            CONSTANT ab,ai, bb,bi : IN SchedType
          ) RETURN SchedType IS
        VARIABLE z : SchedType;
    BEGIN
        -- z = (a AND b) OR (a NOR b)
        z.inp0  :=   Maximum (  Minimum (ab.inp0 , bb.inp0 ),
                                Minimum (ai.inp1 , bi.inp1 ) );
        z.inp1  :=   Minimum (  Maximum (ab.inp1 , bb.inp1 ),
                                Maximum (ai.inp0 , bi.inp0 ) );
        z.InpX  :=   Maximum (  Maximum (ab.InpX , bb.InpX ),
                                Maximum (ai.InpX , bi.InpX ) );
        z.Glch0 :=   Maximum (GlitchMinTime (ab.Glch0, bb.Glch0),
                              GlitchMinTime (ai.Glch1, bi.Glch1) );
        z.Glch1 := GlitchMinTime (  Maximum (ab.Glch1, bb.Glch1),
                                Maximum (ai.Glch0, bi.Glch0) );
        RETURN (z);
    END;

    -- ---------------------------------------------------------------------
    -- Delay Calculation for 3-bit Logical gates.
    -- ---------------------------------------------------------------------
    IMPURE FUNCTION VitalXOR3   (
            CONSTANT ab,ai, bb,bi, cb,ci : IN SchedType )
      RETURN SchedType IS
    BEGIN
        RETURN VitalXOR2 ( VitalXOR2 (ab,ai, bb,bi),
                           VitalXOR2 (ai,ab, bi,bb),
                           cb, ci );
    END;

    IMPURE FUNCTION VitalXNOR3  (
            CONSTANT ab,ai, bb,bi, cb,ci : IN SchedType )
      RETURN SchedType IS
    BEGIN
        RETURN VitalXNOR2 ( VitalXOR2 ( ab,ai, bb,bi ),
                            VitalXOR2 ( ai,ab, bi,bb ),
                            cb, ci );
    END;

    -- ---------------------------------------------------------------------
    -- Delay Calculation for 4-bit Logical gates.
    -- ---------------------------------------------------------------------
    IMPURE FUNCTION VitalXOR4   (
            CONSTANT ab,ai, bb,bi, cb,ci, db,di : IN SchedType )
      RETURN SchedType IS
    BEGIN
        RETURN VitalXOR2 ( VitalXOR2 ( ab,ai, bb,bi ),
                           VitalXOR2 ( ai,ab, bi,bb ),
                           VitalXOR2 ( cb,ci, db,di ),
                           VitalXOR2 ( ci,cb, di,db ) );
    END;

    IMPURE FUNCTION VitalXNOR4  (
            CONSTANT ab,ai, bb,bi, cb,ci, db,di : IN SchedType )
      RETURN SchedType IS
    BEGIN
        RETURN VitalXNOR2 ( VitalXOR2 ( ab,ai, bb,bi ),
                            VitalXOR2 ( ai,ab, bi,bb ),
                            VitalXOR2 ( cb,ci, db,di ),
                            VitalXOR2 ( ci,cb, di,db ) );
    END;

    -- ---------------------------------------------------------------------
    -- Delay Calculation for N-bit Logical gates.
    -- ---------------------------------------------------------------------
    -- Note: index range on datab,datai assumed to be 1 TO length.
    --       This is enforced by internal only usage of this Function
    IMPURE FUNCTION VitalXOR   (
            CONSTANT DataB, DataI : IN SchedArray
          ) RETURN SchedType IS
            CONSTANT Leng : INTEGER := DataB'LENGTH;
    BEGIN
        IF Leng = 2 THEN
            RETURN VitalXOR2 ( DataB(1),DataI(1), DataB(2),DataI(2) );
        ELSE
            RETURN VitalXOR2 ( VitalXOR ( DataB(1 TO Leng-1),
                                          DataI(1 TO Leng-1) ),
                               VitalXOR ( DataI(1 TO Leng-1),
                                          DataB(1 TO Leng-1) ),
                               DataB(Leng),DataI(Leng) );
```

 Copyright © 2001 IEEE. All rights reserved.

```
        END IF;
    END;

    -- Note: index range on datab,datai assumed to be 1 TO length.
    --       This is enforced by internal only usage of this Function
    IMPURE FUNCTION VitalXNOR  (
            CONSTANT DataB, DataI : IN SchedArray
          ) RETURN SchedType IS
            CONSTANT Leng : INTEGER := DataB'LENGTH;
    BEGIN
        IF Leng = 2 THEN
            RETURN VitalXNOR2 ( DataB(1),DataI(1), DataB(2),DataI(2) );
        ELSE
            RETURN VitalXNOR2 ( VitalXOR ( DataB(1 TO Leng-1),
                                           DataI(1 TO Leng-1) ),
                                VitalXOR ( DataI(1 TO Leng-1),
                                           DataB(1 TO Leng-1) ),
                                DataB(Leng),DataI(Leng) );
        END IF;
    END;

    -- ------------------------------------------------------------------------
    -- Multiplexor
    --   MUX   .......... result := data(dselect)
    --   MUX2  .......... 2-input mux; result := data0 when (dselect = '0'),
    --                                           data1 when (dselect = '1'),
    --                      'X' when (dselect = 'X') and (data0 /= data1)
    --   MUX4  .......... 4-input mux; result := data(dselect)
    --   MUX8  .......... 8-input mux; result := data(dselect)
    -- ------------------------------------------------------------------------
    FUNCTION VitalMUX2  (
            CONSTANT d1, d0 : IN SchedType;
            CONSTANT sb, SI : IN SchedType
          ) RETURN SchedType IS
    BEGIN
        RETURN (d1 AND sb) OR (d0 AND (NOT SI) );
    END;
--
    FUNCTION VitalMUX4  (
            CONSTANT Data : IN SchedArray4;
            CONSTANT sb   : IN SchedArray2;
            CONSTANT SI   : IN SchedArray2
          ) RETURN SchedType IS
    BEGIN
        RETURN    (      sb(1)  AND VitalMUX2(Data(3),Data(2), sb(0), SI(0)) )
               OR ( (NOT SI(1)) AND VitalMUX2(Data(1),Data(0), sb(0), SI(0)) );
    END;

    FUNCTION VitalMUX8  (
            CONSTANT Data : IN SchedArray8;
            CONSTANT sb   : IN SchedArray3;
            CONSTANT SI   : IN SchedArray3
          ) RETURN SchedType IS
    BEGIN
        RETURN    ( (    sb(2)) AND VitalMUX4 (Data(7 DOWNTO 4),
                                           sb(1 DOWNTO 0), SI(1 DOWNTO 0) ) )
               OR ( (NOT SI(2)) AND VitalMUX4 (Data(3 DOWNTO 0),
                                           sb(1 DOWNTO 0), SI(1 DOWNTO 0) ) );
    END;
--
    FUNCTION VInterMux   (
            CONSTANT Data : IN SchedArray;
            CONSTANT sb   : IN SchedArray;
            CONSTANT SI   : IN SchedArray
          ) RETURN SchedType IS
        CONSTANT sMsb : INTEGER := sb'LENGTH;
        CONSTANT dMsbHigh : INTEGER := Data'LENGTH;
        CONSTANT dMsbLow  : INTEGER := Data'LENGTH/2;
    BEGIN
        IF sb'LENGTH = 1 THEN
          RETURN VitalMUX2( Data(2), Data(1), sb(1), SI(1) );
        ELSIF sb'LENGTH = 2 THEN
          RETURN VitalMUX4( Data, sb, SI );
        ELSIF sb'LENGTH = 3 THEN
          RETURN VitalMUX8( Data, sb, SI );
        ELSIF sb'LENGTH > 3 THEN
          RETURN ((    sb(sMsb)) AND VInterMux( Data(dMsbLow  DOWNTO  1),
                                                sb(sMsb-1 DOWNTO 1),
                                                SI(sMsb-1 DOWNTO 1) ))
              OR ((NOT SI(sMsb)) AND VInterMux( Data(dMsbHigh DOWNTO dMsbLow+1),
                                                sb(sMsb-1 DOWNTO 1),
```

Copyright © 2001 IEEE. All rights reserved.

```
                                                SI(sMsb-1 DOWNTO 1) ));
        ELSE
          RETURN (0 ns, 0 ns, 0 ns, 0 ns, 0 ns); -- dselect'LENGTH < 1
        END IF;
    END;
--
    FUNCTION VitalMUX   (
            CONSTANT Data : IN SchedArray;
            CONSTANT sb   : IN SchedArray;
            CONSTANT SI   : IN SchedArray
          ) RETURN SchedType IS
        CONSTANT msb : INTEGER := 2**sb'LENGTH;
        VARIABLE    lDat : SchedArray(msb DOWNTO 1);
        ALIAS DataAlias : SchedArray ( Data'LENGTH DOWNTO 1 ) IS Data;
        ALIAS   sbAlias : SchedArray (   sb'LENGTH DOWNTO 1 ) IS sb;
        ALIAS   siAlias : SchedArray (   SI'LENGTH DOWNTO 1 ) IS SI;
    BEGIN
        IF Data'LENGTH <= msb THEN
            FOR i IN Data'LENGTH DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
            FOR i IN msb DOWNTO Data'LENGTH+1 LOOP
                lDat(i) := DefSchedAnd;
            END LOOP;
        ELSE
            FOR i IN msb DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
        END IF;
        RETURN VInterMux( lDat, sbAlias, siAlias );
    END;

    -- ------------------------------------------------------------------------
    -- Decoder
    --          General Algorithm :
    --              (a) Result(...) := '0' when (enable = '0')
    --              (b) Result(data) := '1'; all other subelements = '0'
    --              ... Result array is decending (n-1 downto 0)
    --
    --          DECODERn  .......... n:2**n decoder
    -- ------------------------------------------------------------------------
    FUNCTION VitalDECODER2  (
            CONSTANT DataB  : IN SchedType;
            CONSTANT DataI  : IN SchedType;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        VARIABLE Result : SchedArray2;
    BEGIN
        Result(1) := Enable AND (    DataB);
        Result(0) := Enable AND (NOT DataI);
        RETURN Result;
    END;

    FUNCTION VitalDECODER4  (
            CONSTANT DataB  : IN SchedArray2;
            CONSTANT DataI  : IN SchedArray2;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        VARIABLE Result : SchedArray4;
    BEGIN
        Result(3) := Enable AND (    DataB(1)) AND (    DataB(0));
        Result(2) := Enable AND (    DataB(1)) AND (NOT DataI(0));
        Result(1) := Enable AND (NOT DataI(1)) AND (    DataB(0));
        Result(0) := Enable AND (NOT DataI(1)) AND (NOT DataI(0));
        RETURN Result;
    END;

    FUNCTION VitalDECODER8  (
            CONSTANT DataB  : IN SchedArray3;
            CONSTANT DataI  : IN SchedArray3;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        VARIABLE Result : SchedArray8;
    BEGIN
        Result(7):= Enable AND (    DataB(2))AND(    DataB(1))AND(    DataB(0));
        Result(6):= Enable AND (    DataB(2))AND(    DataB(1))AND(NOT DataI(0));
        Result(5):= Enable AND (    DataB(2))AND(NOT DataI(1))AND(    DataB(0));
        Result(4):= Enable AND (    DataB(2))AND(NOT DataI(1))AND(NOT DataI(0));
        Result(3):= Enable AND (NOT DataI(2))AND(    DataB(1))AND(    DataB(0));
        Result(2):= Enable AND (NOT DataI(2))AND(    DataB(1))AND(NOT DataI(0));
        Result(1):= Enable AND (NOT DataI(2))AND(NOT DataI(1))AND(    DataB(0));
```

 Copyright © 2001 IEEE. All rights reserved.

```
        Result(0):= Enable AND (NOT DataI(2))AND(NOT DataI(1))AND(NOT DataI(0));
        RETURN Result;
    END;


    FUNCTION VitalDECODER   (
            CONSTANT DataB  : IN SchedArray;
            CONSTANT DataI  : IN SchedArray;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        CONSTANT DMsb : INTEGER := DataB'LENGTH - 1;
        ALIAS DataBAlias : SchedArray ( DMsb DOWNTO 0 ) IS DataB;
        ALIAS DataIAlias : SchedArray ( DMsb DOWNTO 0 ) IS DataI;
    BEGIN
        IF DataB'LENGTH = 1 THEN
            RETURN  VitalDECODER2 ( DataBAlias(    0     ),
                                    DataIAlias(    0     ), Enable );
        ELSIF DataB'LENGTH = 2 THEN
            RETURN  VitalDECODER4 ( DataBAlias(1 DOWNTO 0),
                                    DataIAlias(1 DOWNTO 0), Enable );
        ELSIF DataB'LENGTH = 3 THEN
            RETURN  VitalDECODER8 ( DataBAlias(2 DOWNTO 0),
                                    DataIAlias(2 DOWNTO 0), Enable );
        ELSIF DataB'LENGTH > 3 THEN
            RETURN  VitalDECODER  ( DataBAlias(DMsb-1 DOWNTO 0),
                                    DataIAlias(DMsb-1 DOWNTO 0),
                                    Enable AND (    DataBAlias(DMsb)) )
                  & VitalDECODER  ( DataBAlias(DMsb-1 DOWNTO 0),
                                    DataIAlias(DMsb-1 DOWNTO 0),
                                    Enable AND (NOT DataIAlias(DMsb)) );
        ELSE
            RETURN DefSchedArray2;
        END IF;
    END;


--------------------------------------------------------------------------------
-- PRIMITIVES
--------------------------------------------------------------------------------
    -- ------------------------------------------------------------------------
    -- N-bit wide Logical gates.
    -- ------------------------------------------------------------------------
    FUNCTION VitalAND    (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '1';
        FOR i IN Data'RANGE LOOP
            Result := Result AND Data(i);
        END LOOP;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalOR     (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result OR Data(i);
        END LOOP;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalXOR    (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result XOR Data(i);
        END LOOP;
```


Copyright © 2001 IEEE. All rights reserved.

```
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalNAND   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '1';
        FOR i IN Data'RANGE LOOP
            Result := Result AND Data(i);
        END LOOP;
        RETURN ResultMap(NOT Result);
    END;
--
    FUNCTION VitalNOR    (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result OR Data(i);
        END LOOP;
        RETURN ResultMap(NOT Result);
    END;
--
    FUNCTION VitalXNOR   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result XOR Data(i);
        END LOOP;
        RETURN ResultMap(NOT Result);
    END;

    -- ---------------------------------------------------------------------------
    -- Commonly used 2-bit Logical gates.
    -- ---------------------------------------------------------------------------
    FUNCTION VitalAND2   (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a AND b);
    END;
--
    FUNCTION VitalOR2    (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a OR b);
    END;
--
    FUNCTION VitalXOR2   (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a XOR b);
    END;
--
    FUNCTION VitalNAND2  (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
        RETURN ResultMap(a NAND b);
    END;
--
    FUNCTION VitalNOR2   (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a NOR b);
    END;
--
    FUNCTION VitalXNOR2  (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a XOR b));
    END;
--
    -- ------------------------------------------------------------------------
    -- Commonly used 3-bit Logical gates.
    -- ------------------------------------------------------------------------
    FUNCTION VitalAND3   (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a AND b AND c);
    END;
--
    FUNCTION VitalOR3    (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a OR b OR c);
    END;
--
    FUNCTION VitalXOR3   (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a XOR b XOR c);
    END;
--
    FUNCTION VitalNAND3  (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a AND b AND c));
    END;
--
    FUNCTION VitalNOR3   (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a OR b OR c));
    END;
--
    FUNCTION VitalXNOR3  (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
         ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a XOR b XOR c));
    END;

    -- ---------------------------------------------------------------------------
    -- Commonly used 4-bit Logical gates.
    -- ---------------------------------------------------------------------------
```

Copyright © 2001 IEEE. All rights reserved.

```
    FUNCTION VitalAND4   (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a AND b AND c AND d);
    END;
--
    FUNCTION VitalOR4    (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a OR b OR c OR d);
    END;
--
    FUNCTION VitalXOR4   (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a XOR b XOR c XOR d);
    END;
--
    FUNCTION VitalNAND4  (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a AND b AND c AND d));
    END;
--
    FUNCTION VitalNOR4   (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a OR b OR c OR d));
    END;
--
    FUNCTION VitalXNOR4  (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a XOR b XOR c XOR d));
    END;

    -- ---------------------------------------------------------------------------
    -- Buffers
    --   BUF    ....... standard non-inverting buffer
    --   BUFIF0 ....... non-inverting buffer Data passes thru if (Enable = '0')
    --   BUFIF1 ....... non-inverting buffer Data passes thru if (Enable = '1')
    -- ---------------------------------------------------------------------------
    FUNCTION VitalBUF    (
            CONSTANT         Data :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(To_UX01(Data));
    END;
--
    FUNCTION VitalBUFIF0 (
            CONSTANT Data, Enable :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(BufIf0_Table(Enable,Data));
    END;
--
    FUNCTION VitalBUFIF1 (
            CONSTANT Data, Enable :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultZMapType
```

 Copyright © 2001 IEEE. All rights reserved.

```
                                        := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
      BEGIN
          RETURN ResultMap(BufIf1_Table(Enable,Data));
      END;
      FUNCTION VitalIDENT  (
              CONSTANT         Data :  IN std_ulogic;
              CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
      BEGIN
          RETURN ResultMap(To_UX01Z(Data));
      END;

      -- ---------------------------------------------------------------------
      -- Invertors
      --   INV    ......... standard inverting buffer
      --   INVIF0 ......... inverting buffer Data passes thru if (Enable = '0')
      --   INVIF1 ......... inverting buffer Data passes thru if (Enable = '1')
      -- ---------------------------------------------------------------------
      FUNCTION VitalINV    (
              CONSTANT         Data :  IN std_ulogic;
              CONSTANT    ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
          ) RETURN std_ulogic IS
      BEGIN
          RETURN ResultMap(NOT Data);
      END;
  --
      FUNCTION VitalINVIF0 (
              CONSTANT Data, Enable :  IN std_ulogic;
              CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
      BEGIN
          RETURN ResultMap(InvIf0_Table(Enable,Data));
      END;
  --
      FUNCTION VitalINVIF1 (
              CONSTANT Data, Enable :  IN std_ulogic;
              CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
      BEGIN
          RETURN ResultMap(InvIf1_Table(Enable,Data));
      END;

      -- ---------------------------------------------------------------------
      -- Multiplexor
      --   MUX   .......... result := data(dselect)
      --   MUX2  .......... 2-input mux; result := data0 when (dselect = '0'),
      --                                           data1 when (dselect = '1'),
      --                        'X' when (dselect = 'X') and (data0 /= data1)
      --   MUX4  .......... 4-input mux; result := data(dselect)
      --   MUX8  .......... 8-input mux; result := data(dselect)
      -- ---------------------------------------------------------------------
      FUNCTION VitalMUX2  (
              CONSTANT Data1, Data0 :  IN std_ulogic;
              CONSTANT      dSelect :  IN std_ulogic;
              CONSTANT    ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
          ) RETURN std_ulogic IS
          VARIABLE Result : UX01;
      BEGIN
          CASE To_X01(dSelect) IS
            WHEN '0'     => Result := To_UX01(Data0);
            WHEN '1'     => Result := To_UX01(Data1);
            WHEN OTHERS => Result := VitalSame( Data1, Data0 );
          END CASE;
          RETURN ResultMap(Result);
      END;
  --
      FUNCTION VitalMUX4  (
              CONSTANT       Data :  IN std_logic_vector4;
              CONSTANT    dSelect :  IN std_logic_vector2;
              CONSTANT  ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
          ) RETURN std_ulogic IS
          VARIABLE Slct : std_logic_vector2;
          VARIABLE Result : UX01;
      BEGIN
```

Copyright © 2001 IEEE. All rights reserved.

```
        Slct := To_X01(dSelect);
        CASE Slct IS
          WHEN "00"   => Result := To_UX01(Data(0));
          WHEN "01"   => Result := To_UX01(Data(1));
          WHEN "10"   => Result := To_UX01(Data(2));
          WHEN "11"   => Result := To_UX01(Data(3));
          WHEN "0X"   => Result := VitalSame( Data(1), Data(0) );
          WHEN "1X"   => Result := VitalSame( Data(2), Data(3) );
          WHEN "X0"   => Result := VitalSame( Data(2), Data(0) );
          WHEN "X1"   => Result := VitalSame( Data(3), Data(1) );
          WHEN OTHERS => Result := VitalSame( VitalSame(Data(3),Data(2)),
                                              VitalSame(Data(1),Data(0)));
        END CASE;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalMUX8  (
            CONSTANT       Data :  IN std_logic_vector8;
            CONSTANT    dSelect :  IN std_logic_vector3;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        CASE To_X01(dSelect(2)) IS
          WHEN '0'    => Result := VitalMUX4( Data(3 DOWNTO 0),
                                              dSelect(1 DOWNTO 0));
          WHEN '1'    => Result := VitalMUX4( Data(7 DOWNTO 4),
                                              dSelect(1 DOWNTO 0));
          WHEN OTHERS => Result := VitalSame( VitalMUX4( Data(3 DOWNTO 0),
                                                         dSelect(1 DOWNTO 0)),
                                              VitalMUX4( Data(7 DOWNTO 4),
                                                         dSelect(1 DOWNTO 0)));
        END CASE;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VInterMux    (
            CONSTANT Data    : IN std_logic_vector;
            CONSTANT dSelect : IN std_logic_vector
          ) RETURN std_ulogic IS

        CONSTANT sMsb     : INTEGER := dSelect'LENGTH;
        CONSTANT dMsbHigh : INTEGER := Data'LENGTH;
        CONSTANT dMsbLow  : INTEGER := Data'LENGTH/2;
        ALIAS DataAlias : std_logic_vector (   Data'LENGTH DOWNTO 1) IS Data;
        ALIAS dSelAlias : std_logic_vector (dSelect'LENGTH DOWNTO 1) IS dSelect;

        VARIABLE Result : UX01;
    BEGIN
        IF dSelect'LENGTH = 1 THEN
            Result := VitalMUX2( DataAlias(2), DataAlias(1), dSelAlias(1) );
        ELSIF dSelect'LENGTH = 2 THEN
            Result := VitalMUX4( DataAlias, dSelAlias );
        ELSIF dSelect'LENGTH > 2 THEN
          CASE To_X01(dSelect(sMsb)) IS
            WHEN '0'    =>
              Result := VInterMux( DataAlias(dMsbLow  DOWNTO          1),
                                   dSelAlias(sMsb-1 DOWNTO 1) );
            WHEN '1'    =>
              Result := VInterMux( DataAlias(dMsbHigh DOWNTO dMsbLow+1),
                                   dSelAlias(sMsb-1 DOWNTO 1) );
            WHEN OTHERS =>
              Result := VitalSame(
                            VInterMux( DataAlias(dMsbLow  DOWNTO          1),
                                       dSelAlias(sMsb-1 DOWNTO 1) ),
                            VInterMux( DataAlias(dMsbHigh DOWNTO dMsbLow+1),
                                       dSelAlias(sMsb-1 DOWNTO 1) )
                            );
          END CASE;
        ELSE
          Result := 'X'; -- dselect'LENGTH < 1
        END IF;
        RETURN Result;
    END;
--
    FUNCTION VitalMUX   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT    dSelect :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
```

 Copyright © 2001 IEEE. All rights reserved.

```
        ) RETURN std_ulogic IS
        CONSTANT msb    : INTEGER := 2**dSelect'LENGTH;
        ALIAS DataAlias : std_logic_vector (   Data'LENGTH DOWNTO 1) IS Data;
        ALIAS dSelAlias : std_logic_vector (dSelect'LENGTH DOWNTO 1) IS dSelect;
        VARIABLE lDat   : std_logic_vector(msb DOWNTO 1) := (OTHERS=>'X');
        VARIABLE Result : UX01;
    BEGIN
        IF Data'LENGTH <= msb THEN
            FOR i IN Data'LENGTH DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
        ELSE
            FOR i IN msb DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
        END IF;
        Result := VInterMux( lDat, dSelAlias );
        RETURN ResultMap(Result);
    END;

    -- ------------------------------------------------------------------------
    -- Decoder
    --          General Algorithm :
    --              (a) Result(...) := '0' when (enable = '0')
    --              (b) Result(data) := '1'; all other subelements = '0'
    --              ... Result array is decending (n-1 downto 0)
    --
    --          DECODERn  .......... n:2**n decoder
    -- ------------------------------------------------------------------------
    FUNCTION VitalDECODER2  (
            CONSTANT       Data :  IN std_ulogic;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
        ) RETURN std_logic_vector2 IS
        VARIABLE Result : std_logic_vector2;
    BEGIN
        Result(1) := ResultMap(Enable AND (    Data));
        Result(0) := ResultMap(Enable AND (NOT Data));
        RETURN Result;
    END;
--
    FUNCTION VitalDECODER4  (
            CONSTANT       Data :  IN std_logic_vector2;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
        ) RETURN std_logic_vector4 IS
        VARIABLE Result : std_logic_vector4;
    BEGIN
        Result(3) := ResultMap(Enable AND (    Data(1)) AND (    Data(0)));
        Result(2) := ResultMap(Enable AND (    Data(1)) AND (NOT Data(0)));
        Result(1) := ResultMap(Enable AND (NOT Data(1)) AND (    Data(0)));
        Result(0) := ResultMap(Enable AND (NOT Data(1)) AND (NOT Data(0)));
        RETURN Result;
    END;
--
    FUNCTION VitalDECODER8  (
            CONSTANT       Data :  IN std_logic_vector3;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
        ) RETURN std_logic_vector8 IS
        VARIABLE Result : std_logic_vector8;
    BEGIN
        Result(7) := (    Data(2)) AND (    Data(1)) AND (    Data(0));
        Result(6) := (    Data(2)) AND (    Data(1)) AND (NOT Data(0));
        Result(5) := (    Data(2)) AND (NOT Data(1)) AND (    Data(0));
        Result(4) := (    Data(2)) AND (NOT Data(1)) AND (NOT Data(0));
        Result(3) := (NOT Data(2)) AND (    Data(1)) AND (    Data(0));
        Result(2) := (NOT Data(2)) AND (    Data(1)) AND (NOT Data(0));
        Result(1) := (NOT Data(2)) AND (NOT Data(1)) AND (    Data(0));
        Result(0) := (NOT Data(2)) AND (NOT Data(1)) AND (NOT Data(0));

        Result(0) := ResultMap ( Enable AND Result(0) );
        Result(1) := ResultMap ( Enable AND Result(1) );
        Result(2) := ResultMap ( Enable AND Result(2) );
        Result(3) := ResultMap ( Enable AND Result(3) );
        Result(4) := ResultMap ( Enable AND Result(4) );
        Result(5) := ResultMap ( Enable AND Result(5) );
        Result(6) := ResultMap ( Enable AND Result(6) );
```

Copyright © 2001 IEEE. All rights reserved.

```
        Result(7) := ResultMap ( Enable AND Result(7) );

        RETURN Result;
    END;
--
    FUNCTION VitalDECODER   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_logic_vector IS

        CONSTANT DMsb : INTEGER := Data'LENGTH - 1;
        ALIAS DataAlias : std_logic_vector ( DMsb DOWNTO 0 ) IS Data;
    BEGIN
        IF    Data'LENGTH = 1 THEN
            RETURN VitalDECODER2 (DataAlias(    0     ), Enable, ResultMap );
        ELSIF Data'LENGTH = 2 THEN
            RETURN VitalDECODER4 (DataAlias(1 DOWNTO 0), Enable, ResultMap );
        ELSIF Data'LENGTH = 3 THEN
            RETURN VitalDECODER8 (DataAlias(2 DOWNTO 0), Enable, ResultMap );
        ELSIF Data'LENGTH > 3 THEN
            RETURN VitalDECODER  (DataAlias(DMsb-1 DOWNTO 0),
                                  Enable AND (    DataAlias(DMsb)), ResultMap )
                 & VitalDECODER  (DataAlias(DMsb-1 DOWNTO 0),
                                  Enable AND (NOT DataAlias(DMsb)), ResultMap );
        ELSE RETURN "X";
        END IF;
    END;

    -- ------------------------------------------------------------------------
    -- N-bit wide Logical gates.
    -- ------------------------------------------------------------------------
    PROCEDURE VitalAND   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
        VARIABLE Data_Schd : SchedArray(Data'RANGE);
        VARIABLE NewValue     : UX01;
        VARIABLE Glitch_Data : GlitchDataType;
        VARIABLE new_schd    : SchedType;
        VARIABLE Dly, Glch   : TIME;
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalAND(Data, ResultMap);
          WAIT ON Data;
      END LOOP;
      ELSE

        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        FOR n IN Data'RANGE LOOP
            BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        BufPath ( Data_Schd, Data_Edge, Atpd_data_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := '1';
        new_schd := Data_Schd(Data_Schd'LEFT);
        FOR i IN Data'RANGE LOOP
            NewValue  := NewValue  AND Data(i);
            new_schd := new_schd AND Data_Schd(i);
        END LOOP;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data;
      END LOOP;
      END IF; --SN
    END;
--
    PROCEDURE VitalOR    (
            SIGNAL           q : OUT std_ulogic;
            SIGNAL        Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
        VARIABLE Data_Schd : SchedArray(Data'RANGE);
        VARIABLE NewValue     : UX01;
        VARIABLE Glitch_Data : GlitchDataType;
        VARIABLE new_schd    : SchedType;
        VARIABLE Dly, Glch   : TIME;
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalOR(Data, ResultMap);
          WAIT ON Data;
      END LOOP;
      ELSE

        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        FOR n IN Data'RANGE LOOP
            BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := '0';
        new_schd := Data_Schd(Data_Schd'LEFT);
        FOR i IN Data'RANGE LOOP
            NewValue  := NewValue  OR Data(i);
```

Copyright © 2001 IEEE. All rights reserved.

```
                new_schd := new_schd OR Data_Schd(i);
            END LOOP;

            -- ------------------------------------------------------
            -- Assign Outputs
            --  get delays to new value and possable glitch
            --  schedule output change with On Event glitch detection
            -- ------------------------------------------------------
            GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
            VitalGlitchOnEvent ( q, “q”, Glitch_Data, ResultMap(NewValue), Dly,
                                 PrimGlitchMode, GlitchDelay=>Glch );

            WAIT ON Data;
          END LOOP;
          END IF; --SN
        END;
    --
        PROCEDURE VitalXOR   (
                SIGNAL            q : OUT std_ulogic;
                SIGNAL         Data :  IN std_logic_vector;
                CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
                CONSTANT  ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
        ) IS
            VARIABLE LastData  : std_logic_vector(Data’RANGE) := (OTHERS=>’U’);
            VARIABLE Data_Edge   :  EdgeArray(Data’RANGE);
            VARIABLE DataB_Schd  : SchedArray(1 TO Data’LENGTH);
            VARIABLE DataI_Schd  : SchedArray(1 TO Data’LENGTH);
            VARIABLE NewValue     : UX01;
            VARIABLE Glitch_Data : GlitchDataType;
            VARIABLE new_schd    : SchedType;
            VARIABLE Dly, Glch   : TIME;
            ALIAS Atpd_data_q : VitalDelayArrayType01(Data’RANGE) IS tpd_data_q;
            ALIAS ADataB_Schd : SchedArray(Data’RANGE) IS DataB_Schd;
            ALIAS ADataI_Schd : SchedArray(Data’RANGE) IS DataI_Schd;
            VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
        BEGIN
          -- ------------------------------------------------------------------------
          --  Check if ALL zero delay paths, use simple model
          --   ( No delay selection, glitch detection required )
          -- ------------------------------------------------------------------------
          FOR i IN Data’RANGE LOOP
              IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                  AllZeroDelay := FALSE;
                  EXIT;
              END IF;
          END LOOP;
          IF (AllZeroDelay) THEN LOOP
              q <= VitalXOR(Data, ResultMap);
              WAIT ON Data;
          END LOOP;
          ELSE

            -- --------------------------------------
            -- Initialize delay schedules
            -- --------------------------------------
            FOR n IN Data’RANGE LOOP
                BufPath ( ADataB_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
                InvPath ( ADataI_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
            END LOOP;

          LOOP
            -- --------------------------------------
            -- Process input signals
            --   get edge values
            --   re-evaluate output schedules
            -- --------------------------------------
            GetEdge ( Data, LastData, Data_Edge );
            BufPath ( ADataB_Schd, Data_Edge, Atpd_data_q );
            InvPath ( ADataI_Schd, Data_Edge, Atpd_data_q );

            -- ------------------------------------
            -- Compute function and propation delay
            -- ------------------------------------
            NewValue  := VitalXOR ( Data );
            new_schd := VitalXOR ( DataB_Schd, DataI_Schd );

            -- ------------------------------------------------------
            -- Assign Outputs
            --  get delays to new value and possable glitch
            --  schedule output change with On Event glitch detection
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data;
      END LOOP;
      END IF; --SN
    END;
--
    PROCEDURE VitalNAND  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
        VARIABLE Data_Schd : SchedArray(Data'RANGE);
        VARIABLE NewValue     : UX01;
        VARIABLE Glitch_Data : GlitchDataType;
        VARIABLE new_schd    : SchedType;
        VARIABLE Dly, Glch   : TIME;
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalNAND(Data, ResultMap);
          WAIT ON Data;
      END LOOP;
      ELSE

        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        FOR n IN Data'RANGE LOOP
            InvPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        InvPath ( Data_Schd, Data_Edge, Atpd_data_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := '1';
        new_schd := Data_Schd(Data_Schd'LEFT);
        FOR i IN Data'RANGE LOOP
            NewValue  := NewValue  AND Data(i);
            new_schd := new_schd AND Data_Schd(i);
        END LOOP;
        NewValue  := NOT NewValue;
        new_schd := NOT new_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data;
```

Copyright © 2001 IEEE. All rights reserved.

```
          END LOOP;
        END IF;
      END;
--
      PROCEDURE VitalNOR   (
              SIGNAL            q : OUT std_ulogic;
              SIGNAL         Data :  IN std_logic_vector;
              CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
              CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
      ) IS
          VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
          VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
          VARIABLE Data_Schd : SchedArray(Data'RANGE);
          VARIABLE NewValue     : UX01;
          VARIABLE Glitch_Data : GlitchDataType;
          VARIABLE new_schd    : SchedType;
          VARIABLE Dly, Glch   : TIME;
          ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
          VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
      BEGIN
        -- ------------------------------------------------------------------------
        --  Check if ALL zero delay paths, use simple model
        --   ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        FOR i IN Data'RANGE LOOP
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;
        IF (AllZeroDelay) THEN LOOP
            q <= VitalNOR(Data, ResultMap);
            WAIT ON Data;
        END LOOP;
        ELSE

          -- --------------------------------------
          -- Initialize delay schedules
          -- --------------------------------------
          FOR n IN Data'RANGE LOOP
              InvPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
          END LOOP;

        LOOP
          -- --------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- --------------------------------------
          GetEdge ( Data, LastData, Data_Edge );
          InvPath ( Data_Schd, Data_Edge, Atpd_data_q );

          -- ------------------------------------
          -- Compute function and propation delay
          -- ------------------------------------
          NewValue := '0';
          new_schd := Data_Schd(Data_Schd'LEFT);
          FOR i IN Data'RANGE LOOP
              NewValue  := NewValue  OR Data(i);
              new_schd := new_schd OR Data_Schd(i);
          END LOOP;
          NewValue  := NOT NewValue;
          new_schd := NOT new_schd;

          -- ------------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ------------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                               PrimGlitchMode, GlitchDelay=>Glch );

          WAIT ON Data;
        END LOOP;
        END IF; --SN
      END;
--
      PROCEDURE VitalXNOR  (
              SIGNAL            q : OUT std_ulogic;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        SIGNAL         Data :  IN std_logic_vector;
        CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
) IS
    VARIABLE LastData    : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
    VARIABLE Data_Edge   :  EdgeArray(Data'RANGE);
    VARIABLE DataB_Schd  : SchedArray(1 TO Data'LENGTH);
    VARIABLE DataI_Schd  : SchedArray(1 TO Data'LENGTH);
    VARIABLE NewValue     : UX01;
    VARIABLE Glitch_Data : GlitchDataType;
    VARIABLE new_schd    : SchedType;
    VARIABLE Dly, Glch   : TIME;
    ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
    ALIAS ADataB_Schd : SchedArray(Data'RANGE) IS DataB_Schd;
    ALIAS ADataI_Schd : SchedArray(Data'RANGE) IS DataI_Schd;
    VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
BEGIN
  -- ------------------------------------------------------------------------
  --  Check if ALL zero delay paths, use simple model
  --   ( No delay selection, glitch detection required )
  -- ------------------------------------------------------------------------
  FOR i IN Data'RANGE LOOP
      IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
          AllZeroDelay := FALSE;
          EXIT;
      END IF;
  END LOOP;
  IF (AllZeroDelay) THEN LOOP
      q <= VitalXNOR(Data, ResultMap);
      WAIT ON Data;
  END LOOP;
  ELSE

    -- ------------------------------------
    -- Initialize delay schedules
    -- ------------------------------------
    FOR n IN Data'RANGE LOOP
        BufPath ( ADataB_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        InvPath ( ADataI_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
    END LOOP;

  LOOP
    -- ------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- ------------------------------------
    GetEdge ( Data, LastData, Data_Edge );
    BufPath ( ADataB_Schd, Data_Edge, Atpd_data_q );
    InvPath ( ADataI_Schd, Data_Edge, Atpd_data_q );

    -- ----------------------------------
    -- Compute function and propation delay
    -- ----------------------------------
    NewValue  := VitalXNOR ( Data );
    new_schd := VitalXNOR ( DataB_Schd, DataI_Schd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON Data;
  END LOOP;
  END IF; --SN
END;
--

-- ------------------------------------------------------------------------
-- Commonly used 2-bit Logical gates.
-- ------------------------------------------------------------------------
PROCEDURE VitalAND2  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL         a, b :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
```

Copyright © 2001 IEEE. All rights reserved.

```
            CONSTANT  ResultMap :  IN VitalResultMapType
                                     := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalAND2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := a AND b;
        new_schd := a_schd AND b_schd;

        -- ---------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ---------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalOR2   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                     := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalOR2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    ELSE
        -- ------------------------------------
        -- Initialize delay schedules
        -- ------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- ------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );

        -- -----------------------------------
        -- Compute function and propation delay
        -- -----------------------------------
        NewValue := a OR b;
        new_schd := a_schd OR b_schd;

        -- ---------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ---------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNAND2 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNAND2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- ------------------------------------
        -- Initialize delay schedules
        -- ------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- ------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );

        -- -----------------------------------
        -- Compute function and propation delay
        -- -----------------------------------
        NewValue := a NAND b;
```

```
        new_schd := a_schd NAND b_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNOR2  (
            SIGNAL           q : OUT std_ulogic;
            SIGNAL        a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNOR2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := a NOR b;
        new_schd := a_schd NOR b_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXOR2  (
            SIGNAL           q : OUT std_ulogic;
            SIGNAL        a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
```


Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
) IS
    VARIABLE ab_schd, bb_schd : SchedType;
    VARIABLE ai_schd, bi_schd : SchedType;
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
  LOOP
    q <= VitalXOR2 ( a, b, ResultMap );
    WAIT ON a, b;
  END LOOP;

ELSE
    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
    InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
    BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
    InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    BufPath ( ab_schd, GetEdge(a), tpd_a_q );
    InvPath ( ai_schd, GetEdge(a), tpd_a_q );

    BufPath ( bb_schd, GetEdge(b), tpd_b_q );
    InvPath ( bi_schd, GetEdge(b), tpd_b_q );

    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := a XOR b;
    new_schd := VitalXOR2 ( ab_schd,ai_schd, bb_schd,bi_schd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, “q”, Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON a, b;
  END LOOP;
END IF;
END;
--
PROCEDURE VitalXNOR2 (
        SIGNAL           q : OUT std_ulogic;
        SIGNAL        a, b :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
) IS
    VARIABLE ab_schd, bb_schd : SchedType;
    VARIABLE ai_schd, bi_schd : SchedType;
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
```

Copyright © 2001 IEEE. All rights reserved.

```
    --   ( No delay selection, glitch detection required )
    -- -------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXNOR2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( ab_schd, GetEdge(a), tpd_a_q );
        InvPath ( ai_schd, GetEdge(a), tpd_a_q );

        BufPath ( bb_schd, GetEdge(b), tpd_b_q );
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := NOT (a XOR b);
        new_schd := VitalXNOR2 ( ab_schd,ai_schd, bb_schd,bi_schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;

    -- -------------------------------------------------------------------------
    -- Commonly used 3-bit Logical gates.
    -- -------------------------------------------------------------------------
    PROCEDURE VitalAND3  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN
--
    -- -------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- -------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalAND3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------
        -- Initialize delay schedules
        -- ------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );
        BufPath ( c_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- ------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );
        BufPath ( c_schd, GetEdge(c), tpd_c_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a AND b AND c;
        new_schd := a_schd AND b_schd AND c_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalOR3   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ----------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ----------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalOR3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- ------------------------------------
        -- Initialize delay schedules
        -- ------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );
        BufPath ( c_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- ------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );
```

Copyright © 2001 IEEE. All rights reserved.

```
        BufPath ( c_schd, GetEdge(c), tpd_c_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a OR b OR c;
        new_schd := a_schd OR b_schd OR c_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNAND3 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNAND3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );
        InvPath ( c_schd, GetEdge(c), tpd_c_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := (a AND b) NAND c;
        new_schd := (a_schd AND b_schd) NAND c_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
```

 Copyright © 2001 IEEE. All rights reserved.

```
                         PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNOR3  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNOR3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );
        InvPath ( c_schd, GetEdge(c), tpd_c_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := (a OR b) NOR c;
        new_schd := (a_schd OR b_schd) NOR c_schd;

        -- ----------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ----------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXOR3  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
```

Copyright © 2001 IEEE. All rights reserved.

```
    ) IS
        VARIABLE ab_schd, bb_schd, cb_schd : SchedType;
        VARIABLE ai_schd, bi_schd, ci_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXOR3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );
        BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
        InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( ab_schd, GetEdge(a), tpd_a_q );
        InvPath ( ai_schd, GetEdge(a), tpd_a_q );

        BufPath ( bb_schd, GetEdge(b), tpd_b_q );
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );

        BufPath ( cb_schd, GetEdge(c), tpd_c_q );
        InvPath ( ci_schd, GetEdge(c), tpd_c_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a XOR b XOR c;
        new_schd := VitalXOR3 ( ab_schd,ai_schd,
                                bb_schd,bi_schd,
                                cb_schd,ci_schd );

        -- --------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- --------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXNOR3 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd, cb_schd : SchedType;
        VARIABLE ai_schd, bi_schd, ci_schd : SchedType;
        VARIABLE NewValue        : UX01;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
BEGIN

-- ---------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ---------------------------------------------------------------------------
IF (     (tpd_a_q = VitalZeroDelay01)
     AND (tpd_b_q = VitalZeroDelay01)
     AND (tpd_c_q = VitalZeroDelay01)) THEN
  LOOP
    q <= VitalXNOR3 ( a, b, c, ResultMap );
    WAIT ON a, b, c;
  END LOOP;

ELSE
    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
    InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
    BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
    InvPath ( bi_schd, InitialEdge(b), tpd_b_q );
    BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
    InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    BufPath ( ab_schd, GetEdge(a), tpd_a_q );
    InvPath ( ai_schd, GetEdge(a), tpd_a_q );

    BufPath ( bb_schd, GetEdge(b), tpd_b_q );
    InvPath ( bi_schd, GetEdge(b), tpd_b_q );

    BufPath ( cb_schd, GetEdge(c), tpd_c_q );
    InvPath ( ci_schd, GetEdge(c), tpd_c_q );

    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := NOT (a XOR b XOR c);
    new_schd := VitalXNOR3 ( ab_schd, ai_schd,
                             bb_schd, bi_schd,
                             cb_schd, ci_schd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON a, b, c;
  END LOOP;
END IF;
END;

-- ---------------------------------------------------------------------------
-- Commonly used 4-bit Logical gates.
-- ---------------------------------------------------------------------------
PROCEDURE VitalAND4  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
) IS
    VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
```

Copyright © 2001 IEEE. All rights reserved.

```
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalAND4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );
        BufPath ( c_schd, InitialEdge(c), tpd_c_q );
        BufPath ( d_Schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );
        BufPath ( c_schd, GetEdge(c), tpd_c_q );
        BufPath ( d_Schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a AND b AND c AND d;
        new_schd := a_schd AND b_schd AND c_schd AND d_Schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalOR4   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
```

 Copyright © 2001 IEEE. All rights reserved.

```
             AND (tpd_d_q = VitalZeroDelay01)) THEN
          LOOP
            q <= VitalOR4 ( a, b, c, d, ResultMap );
            WAIT ON a, b, c, d;
          END LOOP;

        ELSE
            -- -------------------------------------
            -- Initialize delay schedules
            -- -------------------------------------
            BufPath ( a_schd, InitialEdge(a), tpd_a_q );
            BufPath ( b_schd, InitialEdge(b), tpd_b_q );
            BufPath ( c_schd, InitialEdge(c), tpd_c_q );
            BufPath ( d_Schd, InitialEdge(d), tpd_d_q );

          LOOP
            -- -------------------------------------
            -- Process input signals
            --   get edge values
            --   re-evaluate output schedules
            -- -------------------------------------
            BufPath ( a_schd, GetEdge(a), tpd_a_q );
            BufPath ( b_schd, GetEdge(b), tpd_b_q );
            BufPath ( c_schd, GetEdge(c), tpd_c_q );
            BufPath ( d_Schd, GetEdge(d), tpd_d_q );

            -- ------------------------------------
            -- Compute function and propation delay
            -- ------------------------------------
            NewValue  := a OR b OR c OR d;
            new_schd := a_schd OR b_schd OR c_schd OR d_Schd;

            -- ------------------------------------------------------
            -- Assign Outputs
            --  get delays to new value and possable glitch
            --  schedule output change with On Event glitch detection
            -- ------------------------------------------------------
            GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
            VitalGlitchOnEvent ( q, “q”, Glitch_Data, ResultMap(NewValue), Dly,
                                 PrimGlitchMode, GlitchDelay=>Glch );

            WAIT ON a, b, c, d;
          END LOOP;
        END IF;
        END;
    --
        PROCEDURE VitalNAND4 (
                SIGNAL            q : OUT std_ulogic;
                SIGNAL   a, b, c, d :  IN std_ulogic   ;
                CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT  ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
        ) IS
            VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
            VARIABLE NewValue         : UX01;
            VARIABLE Glitch_Data    : GlitchDataType;
            VARIABLE new_schd       : SchedType;
            VARIABLE Dly, Glch      : TIME;
        BEGIN

        -- ------------------------------------------------------------------------
        --  For ALL zero delay paths, use simple model
        --   ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        IF (     (tpd_a_q = VitalZeroDelay01)
             AND (tpd_b_q = VitalZeroDelay01)
             AND (tpd_c_q = VitalZeroDelay01)
             AND (tpd_d_q = VitalZeroDelay01)) THEN
          LOOP
            q <= VitalNAND4 ( a, b, c, d, ResultMap );
            WAIT ON a, b, c, d;
          END LOOP;

        ELSE
            -- -------------------------------------
            -- Initialize delay schedules
            -- -------------------------------------
            InvPath ( a_schd, InitialEdge(a), tpd_a_q );
```

Copyright © 2001 IEEE. All rights reserved.

```
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );
        InvPath ( d_Schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- ----------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ----------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );
        InvPath ( c_schd, GetEdge(c), tpd_c_q );
        InvPath ( d_Schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := (a AND b) NAND (c AND d);
        new_schd := (a_schd AND b_schd) NAND (c_schd AND d_Schd);

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNOR4  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data     : GlitchDataType;
        VARIABLE new_schd        : SchedType;
        VARIABLE Dly, Glch       : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNOR4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- ----------------------------------------
        -- Initialize delay schedules
        -- ----------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );
        InvPath ( d_Schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- ----------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ----------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
        InvPath ( b_schd, GetEdge(b), tpd_b_q );
        InvPath ( c_schd, GetEdge(c), tpd_c_q );
        InvPath ( d_Schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := (a OR b) NOR (c OR d);
        new_schd := (a_schd OR b_schd) NOR (c_schd OR d_Schd);

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXOR4  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd, cb_schd, DB_Schd : SchedType;
        VARIABLE ai_schd, bi_schd, ci_schd, di_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXOR4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );

        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

        BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
        InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

        BufPath ( DB_Schd, InitialEdge(d), tpd_d_q );
        InvPath ( di_schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( ab_schd, GetEdge(a), tpd_a_q );
        InvPath ( ai_schd, GetEdge(a), tpd_a_q );

        BufPath ( bb_schd, GetEdge(b), tpd_b_q );
```

Copyright © 2001 IEEE. All rights reserved.

```
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );

        BufPath ( cb_schd, GetEdge(c), tpd_c_q );
        InvPath ( ci_schd, GetEdge(c), tpd_c_q );

        BufPath ( DB_Schd, GetEdge(d), tpd_d_q );
        InvPath ( di_schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a XOR b XOR c XOR d;
        new_schd := VitalXOR4 ( ab_schd,ai_schd, bb_schd,bi_schd,
                                cb_schd,ci_schd, DB_Schd,di_schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXNOR4 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd, cb_schd, DB_Schd : SchedType;
        VARIABLE ai_schd, bi_schd, ci_schd, di_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXNOR4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );

        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

        BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
        InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

        BufPath ( DB_Schd, InitialEdge(d), tpd_d_q );
        InvPath ( di_schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
```

 Copyright © 2001 IEEE. All rights reserved.

```
    -- --------------------------------------
    BufPath ( ab_schd, GetEdge(a), tpd_a_q );
    InvPath ( ai_schd, GetEdge(a), tpd_a_q );

    BufPath ( bb_schd, GetEdge(b), tpd_b_q );
    InvPath ( bi_schd, GetEdge(b), tpd_b_q );

    BufPath ( cb_schd, GetEdge(c), tpd_c_q );
    InvPath ( ci_schd, GetEdge(c), tpd_c_q );

    BufPath ( DB_Schd, GetEdge(d), tpd_d_q );
    InvPath ( di_schd, GetEdge(d), tpd_d_q );

    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := NOT (a XOR b XOR c XOR d);
    new_schd := VitalXNOR4 ( ab_schd,ai_schd, bb_schd,bi_schd,
                             cb_schd,ci_schd, DB_Schd,di_schd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON a, b, c, d;
  END LOOP;
END IF;
END;

-- ------------------------------------------------------------------------
-- Buffers
--   BUF    ....... standard non-inverting buffer
--   BUFIF0 ....... non-inverting buffer Data passes thru if (Enable = '0')
--   BUFIF1 ....... non-inverting buffer Data passes thru if (Enable = '1')
-- ------------------------------------------------------------------------
PROCEDURE VitalBUF   (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL            a :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
) IS
    VARIABLE NewValue      : UX01;
    VARIABLE Glitch_Data  : GlitchDataType;
    VARIABLE Dly, Glch    : TIME;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF (tpd_a_q = VitalZeroDelay01) THEN
  LOOP
    q <= ResultMap(To_UX01(a));
    WAIT ON a;
  END LOOP;

ELSE
  LOOP
    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := To_UX01(a);     -- convert to forcing strengths
    CASE EdgeType'(GetEdge(a)) IS
      WHEN '1'|'/'|'R'|'r'  => Dly := tpd_a_q(tr01);
      WHEN '0'|'\'|'F'|'f'  => Dly := tpd_a_q(tr10);
      WHEN OTHERS           => Dly := Minimum (tpd_a_q(tr01), tpd_a_q(tr10));
    END CASE;

    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode );

    WAIT ON a;
  END LOOP;
END IF;
END;
```

Copyright © 2001 IEEE. All rights reserved.

```
--
    PROCEDURE VitalBUFIF1 (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
    ) IS
        VARIABLE NewValue        : UX01Z;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_data_q   = VitalZeroDelay01 )
         AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
      LOOP
        q <= VitalBUFIF1( Data, Enable, ResultMap );
        WAIT ON Data, Enable;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( d_Schd, InitialEdge(Data), tpd_data_q );
        BufEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( d_Schd, GetEdge(Data), tpd_data_q );
        BufEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := VitalBUFIF1( Data, Enable );

        -- ---------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ---------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                        d_Schd, e1_Schd, e0_Schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalBUFIF0 (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
    ) IS
        VARIABLE NewValue        : UX01Z;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
        VARIABLE ne1_schd, ne0_schd : SchedType;
        VARIABLE Dly, Glch      : TIME;
  BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
```

 Copyright © 2001 IEEE. All rights reserved.

```
--   ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF (     (tpd_data_q   = VitalZeroDelay01 )
     AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
  LOOP
    q <= VitalBUFIF0( Data, Enable, ResultMap );
    WAIT ON Data, Enable;
  END LOOP;

ELSE
    -- ------------------------------------
    -- Initialize delay schedules
    -- ------------------------------------
    BufPath ( d_Schd, InitialEdge(Data), tpd_data_q );
    InvEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

  LOOP
    -- ------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- ------------------------------------
    BufPath ( d_Schd, GetEdge(Data), tpd_data_q );
    InvEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );

    -- ----------------------------------
    -- Compute function and propation delay
    -- ----------------------------------
    NewValue := VitalBUFIF0( Data, Enable );
    ne1_schd := NOT e1_Schd;
    ne0_schd := NOT e0_Schd;

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                    d_Schd, ne1_schd, ne0_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON Data, Enable;
  END LOOP;
END IF;
END;

PROCEDURE VitalIDENT (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL            a :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
        CONSTANT  ResultMap :  IN VitalResultZMapType
                                  := VitalDefaultResultZMap
) IS
    SUBTYPE v2 IS std_logic_vector(0 TO 1);
    VARIABLE NewValue      : UX01Z;
    VARIABLE Glitch_Data  : GlitchDataType;
    VARIABLE Dly, Glch    : TIME;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF (tpd_a_q = VitalZeroDelay01Z) THEN
  LOOP
    q <= ResultMap(To_UX01Z(a));
    WAIT ON a;
  END LOOP;

ELSE
  LOOP
    -- ----------------------------------
    -- Compute function and propation delay
    -- ----------------------------------
    CASE v2'(To_X01Z(NewValue) & To_X01Z(a)) IS
      WHEN "00"   => Dly := tpd_a_q(tr10);
      WHEN "01"   => Dly := tpd_a_q(tr01);
      WHEN "0Z"   => Dly := tpd_a_q(tr0z);
      WHEN "0X"   => Dly := tpd_a_q(tr01);
      WHEN "10"   => Dly := tpd_a_q(tr10);
```

Copyright © 2001 IEEE. All rights reserved.

```
          WHEN "11"   => Dly := tpd_a_q(tr01);
          WHEN "1Z"   => Dly := tpd_a_q(tr1z);
          WHEN "1X"   => Dly := tpd_a_q(tr10);
          WHEN "Z0"   => Dly := tpd_a_q(trz0);
          WHEN "Z1"   => Dly := tpd_a_q(trz1);
          WHEN "ZZ"   => Dly := 0 ns;
          WHEN "ZX"   => Dly := Minimum (tpd_a_q(trz1), tpd_a_q(trz0));
          WHEN "X0"   => Dly := tpd_a_q(tr10);
          WHEN "X1"   => Dly := tpd_a_q(tr01);
          WHEN "XZ"   => Dly := Minimum (tpd_a_q(tr0z), tpd_a_q(tr1z));
          WHEN OTHERS => Dly := Minimum (tpd_a_q(tr01), tpd_a_q(tr10));
        END CASE;
        NewValue  := To_UX01Z(a);

        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode );

        WAIT ON a;
      END LOOP;
    END IF;
    END;

    -- ------------------------------------------------------------------------
    -- Invertors
    --   INV    ......... standard inverting buffer
    --   INVIF0 ......... inverting buffer Data passes thru if (Enable = '0')
    --   INVIF1 ......... inverting buffer Data passes thru if (Enable = '1')
    -- ------------------------------------------------------------------------
    PROCEDURE VitalINV   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL            a :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE NewValue      : UX01;
        VARIABLE Glitch_Data  : GlitchDataType;
        VARIABLE new_schd     : SchedType;
        VARIABLE Dly, Glch    : TIME;
    BEGIN
    IF (tpd_a_q = VitalZeroDelay01) THEN
      LOOP
        q <= ResultMap(NOT a);
        WAIT ON a;
      END LOOP;

    ELSE
      LOOP
        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := NOT a;
        CASE EdgeType'(GetEdge(a)) IS
          WHEN '1'|'/'|'R'|'r'  => Dly := tpd_a_q(tr10);
          WHEN '0'|'\'|'F'|'f'  => Dly := tpd_a_q(tr01);
          WHEN OTHERS           => Dly := Minimum (tpd_a_q(tr01), tpd_a_q(tr10));
        END CASE;

        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode );

        WAIT ON a;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalINVIF1 (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap
    ) IS
        VARIABLE NewValue        : UX01Z;
        VARIABLE new_schd       : SchedType;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
        VARIABLE Dly, Glch      : TIME;
  BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_data_q   = VitalZeroDelay01 )
         AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
      LOOP
        q <= VitalINVIF1( Data, Enable, ResultMap );
        WAIT ON Data, Enable;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        InvPath ( d_Schd, InitialEdge(Data), tpd_data_q );
        BufEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        InvPath ( d_Schd, GetEdge(Data), tpd_data_q );
        BufEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := VitalINVIF1( Data, Enable );
        new_schd := NOT d_Schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                        new_schd, e1_Schd, e0_Schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalINVIF0 (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
    ) IS
        VARIABLE NewValue        : UX01Z;
        VARIABLE new_schd       : SchedType;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
        VARIABLE ne1_schd, ne0_schd : SchedType := DefSchedType;
        VARIABLE Dly, Glch      : TIME;
  BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_data_q   = VitalZeroDelay01 )
         AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
      LOOP
        q <= VitalINVIF0( Data, Enable, ResultMap );
        WAIT ON Data, Enable;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
```

Copyright © 2001 IEEE. All rights reserved.

```
    InvPath ( d_Schd, InitialEdge(Data), tpd_data_q );
    InvEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

  LOOP
    -- -------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- -------------------------------------
    InvPath ( d_Schd, GetEdge(Data), tpd_data_q );
    InvEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );

    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue := VitalINVIF0( Data, Enable );
    ne1_schd := NOT e1_Schd;
    ne0_schd := NOT e0_Schd;
    new_schd := NOT d_Schd;

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                    new_schd, ne1_schd, ne0_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON Data, Enable;
  END LOOP;
END IF;
END;

-- ------------------------------------------------------------------------
-- Multiplexor
--   MUX   .......... result := data(dselect)
--   MUX2  .......... 2-input mux; result := data0 when (dselect = '0'),
--                                           data1 when (dselect = '1'),
--                        'X' when (dselect = 'X') and (data0 /= data1)
--   MUX4  .......... 4-input mux; result := data(dselect)
--   MUX8  .......... 8-input mux; result := data(dselect)
-- ------------------------------------------------------------------------
PROCEDURE VitalMUX2  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL       d1, d0 :  IN std_ulogic;
        SIGNAL         dSel :  IN std_ulogic;
        CONSTANT   tpd_d1_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT   tpd_d0_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_dsel_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
) IS
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
    VARIABLE d1_Schd,  d0_Schd  : SchedType;
    VARIABLE dSel_bSchd, dSel_iSchd : SchedType;
    VARIABLE d1_Edge, d0_Edge, dSel_Edge : EdgeType;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF (     (tpd_d1_q   = VitalZeroDelay01)
     AND (tpd_d0_q   = VitalZeroDelay01)
     AND (tpd_dsel_q = VitalZeroDelay01) ) THEN
  LOOP
    q <= VitalMUX2 ( d1, d0, dSel, ResultMap );
    WAIT ON d1, d0, dSel;
  END LOOP;

ELSE
    -- -------------------------------------
    -- Initialize delay schedules
    -- -------------------------------------
    BufPath ( d1_Schd, InitialEdge(d1), tpd_d1_q );
    BufPath ( d0_Schd, InitialEdge(d0), tpd_d0_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
          BufPath ( dSel_bSchd, InitialEdge(dSel), tpd_dsel_q );
          InvPath ( dSel_iSchd, InitialEdge(dSel), tpd_dsel_q );

        LOOP
          -- --------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- --------------------------------------
          BufPath ( d1_Schd, GetEdge(d1), tpd_d1_q );
          BufPath ( d0_Schd, GetEdge(d0), tpd_d0_q );
          BufPath ( dSel_bSchd, GetEdge(dSel), tpd_dsel_q );
          InvPath ( dSel_iSchd, GetEdge(dSel), tpd_dsel_q );

          -- ------------------------------------
          -- Compute function and propation delaq
          -- ------------------------------------
          NewValue  := VitalMUX2 ( d1, d0, dSel );
          new_schd := VitalMUX2 ( d1_Schd, d0_Schd, dSel_bSchd, dSel_iSchd );

          -- ------------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ------------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                               PrimGlitchMode, GlitchDelay=>Glch );

          WAIT ON d1, d0, dSel;
        END LOOP;
      END IF;
      END;
--
      PROCEDURE VitalMUX4  (
              SIGNAL            q : OUT std_ulogic;
              SIGNAL         Data :  IN std_logic_vector4;
              SIGNAL         dSel :  IN std_logic_vector2;
              CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
              CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
              CONSTANT  ResultMap :  IN VitalResultMapType
                                         := VitalDefaultResultMap
      ) IS
          VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
          VARIABLE LastdSel  : std_logic_vector(dSel'RANGE) := (OTHERS=>'U');
          VARIABLE NewValue        : UX01;
          VARIABLE Glitch_Data    : GlitchDataType;
          VARIABLE new_schd       : SchedType;
          VARIABLE Dly, Glch      : TIME;
          VARIABLE Data_Schd      : SchedArray4;
          VARIABLE Data_Edge      : EdgeArray4;
          VARIABLE dSel_Edge      : EdgeArray2;
          VARIABLE dSel_bSchd     : SchedArray2;
          VARIABLE dSel_iSchd     : SchedArray2;
          ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
          ALIAS Atpd_dsel_q : VitalDelayArrayType01(dSel'RANGE) IS tpd_dsel_q;
          VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
      BEGIN
        -- ------------------------------------------------------------------------
        --  Check if ALL zero delay paths, use simple model
        --    ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        FOR i IN dSel'RANGE LOOP
            IF (Atpd_dsel_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;
        IF (AllZeroDelay) THEN
            FOR i IN Data'RANGE LOOP
                IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                    AllZeroDelay := FALSE;
                    EXIT;
                END IF;
            END LOOP;

            IF (AllZeroDelay) THEN LOOP
                q <= VitalMUX(Data, dSel, ResultMap);
                WAIT ON Data, dSel;
            END LOOP;
            END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
  ELSE

    -- ------------------------------------
    -- Initialize delay schedules
    -- ------------------------------------
    FOR n IN Data'RANGE LOOP
        BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
    END LOOP;
    FOR n IN dSel'RANGE LOOP
        BufPath ( dSel_bSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
        InvPath ( dSel_iSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
    END LOOP;

  LOOP
    -- ------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- ------------------------------------
    GetEdge ( Data, LastData, Data_Edge );
    BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

    GetEdge ( dSel, LastdSel, dSel_Edge );
    BufPath ( dSel_bSchd, dSel_Edge, Atpd_dsel_q );
    InvPath ( dSel_iSchd, dSel_Edge, Atpd_dsel_q );

    -- ------------------------------------
    -- Compute function and propation delaq
    -- ------------------------------------
    NewValue  := VitalMUX4 ( Data, dSel );
    new_schd := VitalMUX4 ( Data_Schd, dSel_bSchd, dSel_iSchd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON Data, dSel;
  END LOOP;
  END IF; --SN
END;

PROCEDURE VitalMUX8  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL         Data :  IN std_logic_vector8;
        SIGNAL         dSel :  IN std_logic_vector3;
        CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
        CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
        CONSTANT  ResultMap :  IN VitalResultMapType  := VitalDefaultResultMap
) IS
    VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
    VARIABLE LastdSel  : std_logic_vector(dSel'RANGE) := (OTHERS=>'U');
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
    VARIABLE Data_Schd      : SchedArray8;
    VARIABLE Data_Edge      : EdgeArray8;
    VARIABLE dSel_Edge      : EdgeArray3;
    VARIABLE dSel_bSchd     : SchedArray3;
    VARIABLE dSel_iSchd     : SchedArray3;
    ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
    ALIAS Atpd_dsel_q : VitalDelayArrayType01(dSel'RANGE) IS tpd_dsel_q;
    VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
BEGIN
  -- ------------------------------------------------------------------------
  --  Check if ALL zero delay paths, use simple model
  --    ( No delay selection, glitch detection required )
  -- ------------------------------------------------------------------------
  FOR i IN dSel'RANGE LOOP
      IF (Atpd_dsel_q(i) /= VitalZeroDelay01) THEN
          AllZeroDelay := FALSE;
          EXIT;
      END IF;
  END LOOP;
  IF (AllZeroDelay) THEN
      FOR i IN Data'RANGE LOOP
```

 Copyright © 2001 IEEE. All rights reserved.

```
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;

        IF (AllZeroDelay) THEN LOOP
            q <= VitalMUX(Data, dSel, ResultMap);
            WAIT ON Data, dSel;
        END LOOP;
        END IF;
     ELSE

       -- -------------------------------------
       -- Initialize delay schedules
       -- -------------------------------------
       FOR n IN Data'RANGE LOOP
           BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
       END LOOP;
       FOR n IN dSel'RANGE LOOP
           BufPath ( dSel_bSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
           InvPath ( dSel_iSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
       END LOOP;

     LOOP
       -- -------------------------------------
       -- Process input signals
       --   get edge values
       --   re-evaluate output schedules
       -- -------------------------------------
       GetEdge ( Data, LastData, Data_Edge );
       BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

       GetEdge ( dSel, LastdSel, dSel_Edge );
       BufPath ( dSel_bSchd, dSel_Edge, Atpd_dsel_q );
       InvPath ( dSel_iSchd, dSel_Edge, Atpd_dsel_q );

       -- ------------------------------------
       -- Compute function and propation delaq
       -- ------------------------------------
       NewValue  := VitalMUX8 ( Data, dSel );
       new_schd := VitalMUX8 ( Data_Schd, dSel_bSchd, dSel_iSchd );

       -- ------------------------------------------------------
       -- Assign Outputs
       --  get delays to new value and possable glitch
       --  schedule output change with On Event glitch detection
       -- ------------------------------------------------------
       GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
       VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                            PrimGlitchMode, GlitchDelay=>Glch );

       WAIT ON Data, dSel;
     END LOOP;
     END IF;
   END;
--
   PROCEDURE VitalMUX   (
           SIGNAL            q : OUT std_ulogic;
           SIGNAL         Data :  IN std_logic_vector;
           SIGNAL         dSel :  IN std_logic_vector;
           CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
           CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
           CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
   ) IS
       VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
       VARIABLE LastdSel  : std_logic_vector(dSel'RANGE) := (OTHERS=>'U');
       VARIABLE NewValue        : UX01;
       VARIABLE Glitch_Data    : GlitchDataType;
       VARIABLE new_schd       : SchedType;
       VARIABLE Dly, Glch      : TIME;
       VARIABLE Data_Schd      : SchedArray(Data'RANGE);
       VARIABLE Data_Edge      : EdgeArray(Data'RANGE);
       VARIABLE dSel_Edge      : EdgeArray(dSel'RANGE);
       VARIABLE dSel_bSchd     : SchedArray(dSel'RANGE);
       VARIABLE dSel_iSchd     : SchedArray(dSel'RANGE);
       ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
       ALIAS Atpd_dsel_q : VitalDelayArrayType01(dSel'RANGE) IS tpd_dsel_q;
       VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
   BEGIN
```

Copyright © 2001 IEEE. All rights reserved.

```
    -- ---------------------------------------------------------------------------
    --  Check if ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    FOR i IN dSel'RANGE LOOP
        IF (Atpd_dsel_q(i) /= VitalZeroDelay01) THEN
            AllZeroDelay := FALSE;
            EXIT;
        END IF;
    END LOOP;
    IF (AllZeroDelay) THEN
        FOR i IN Data'RANGE LOOP
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;

        IF (AllZeroDelay) THEN LOOP
            q <= VitalMUX(Data, dSel, ResultMap);
            WAIT ON Data, dSel;
        END LOOP;
        END IF;
    ELSE

      -- --------------------------------------
      -- Initialize delay schedules
      -- --------------------------------------
      FOR n IN Data'RANGE LOOP
          BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
      END LOOP;
      FOR n IN dSel'RANGE LOOP
          BufPath ( dSel_bSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
          InvPath ( dSel_iSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
      END LOOP;

    LOOP
      -- --------------------------------------
      -- Process input signals
      --   get edge values
      --   re-evaluate output schedules
      -- --------------------------------------
      GetEdge ( Data, LastData, Data_Edge );
      BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

      GetEdge ( dSel, LastdSel, dSel_Edge );
      BufPath ( dSel_bSchd, dSel_Edge, Atpd_dsel_q );
      InvPath ( dSel_iSchd, dSel_Edge, Atpd_dsel_q );

      -- ------------------------------------
      -- Compute function and propation delaq
      -- ------------------------------------
      NewValue  := VitalMUX ( Data, dSel );
      new_schd := VitalMUX ( Data_Schd, dSel_bSchd, dSel_iSchd );

      -- -----------------------------------------------------
      -- Assign Outputs
      --  get delays to new value and possable glitch
      --  schedule output change with On Event glitch detection
      -- -----------------------------------------------------
      GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
      VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                           PrimGlitchMode, GlitchDelay=>Glch );

      WAIT ON Data, dSel;
    END LOOP;
   END IF; --SN
  END;

  -- ---------------------------------------------------------------------------
  -- Decoder
  --         General Algorithm :
  --             (a) Result(...) := '0' when (enable = '0')
  --             (b) Result(data) := '1'; all other subelements = '0'
  --             ... Result array is decending (n-1 downto 0)
  --
  --         DECODERn  .......... n:2**n decoder
  -- Caution: If 'ResultMap' defines other than strength mapping, the
  --         delay selection is not defined.
  -- ---------------------------------------------------------------------------
  PROCEDURE VitalDECODER2  (
```

 Copyright © 2001 IEEE. All rights reserved.

```
            SIGNAL             q : OUT std_logic_vector2;
            SIGNAL          Data :  IN std_ulogic;
            SIGNAL        Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
    ) IS
        VARIABLE NewValue        : std_logic_vector2;
        VARIABLE Glitch_Data    : GlitchArray2;
        VARIABLE new_schd       : SchedArray2;
        VARIABLE Dly, Glch      : TimeArray2;
        VARIABLE Enable_Schd  : SchedType := DefSchedType;
        VARIABLE Data_BSchd, Data_ISchd : SchedType;
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      IF (tpd_enable_q = VitalZeroDelay01) AND (tpd_data_q = VitalZeroDelay01) THEN
      LOOP
          q <= VitalDECODER2(Data, Enable, ResultMap);
          WAIT ON Data, Enable;
      END LOOP;
      ELSE

        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( Data_BSchd, InitialEdge(Data), tpd_data_q );
        InvPath ( Data_ISchd, InitialEdge(Data), tpd_data_q );
        BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( Data_BSchd, GetEdge(Data), tpd_data_q );
        InvPath ( Data_ISchd, GetEdge(Data), tpd_data_q );

        BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

        -- ------------------------------------
        -- Compute function and propation delaq
        -- ------------------------------------
        NewValue  := VitalDECODER2 ( Data, Enable, ResultMap );
        new_schd := VitalDECODER2 ( Data_BSchd, Data_ISchd, Enable_Schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
      END IF; -- SN
    END;
--
    PROCEDURE VitalDECODER4  (
            SIGNAL             q : OUT std_logic_vector4;
            SIGNAL          Data :  IN std_logic_vector2;
            SIGNAL        Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
           CONSTANT    ResultMap :  IN VitalResultMapType      := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE NewValue        : std_logic_vector4;
        VARIABLE Glitch_Data    : GlitchArray4;
        VARIABLE new_schd       : SchedArray4;
        VARIABLE Dly, Glch      : TimeArray4;
        VARIABLE Enable_Schd    : SchedType;
        VARIABLE Enable_Edge    : EdgeType;
        VARIABLE Data_Edge      : EdgeArray2;
        VARIABLE Data_BSchd, Data_ISchd : SchedArray2;
```

Copyright © 2001 IEEE. All rights reserved.

```
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ---------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --    ( No delay selection, glitch detection required )
      -- ---------------------------------------------------------------------
      IF (tpd_enable_q /= VitalZeroDelay01) THEN
          AllZeroDelay := FALSE;
      ELSE
          FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
          END LOOP;
      END IF;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalDECODER4(Data, Enable, ResultMap);
          WAIT ON Data, Enable;
      END LOOP;
      ELSE

        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        FOR n IN Data'RANGE LOOP
            BufPath ( Data_BSchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
            InvPath ( Data_ISchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;
        BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        BufPath ( Data_BSchd, Data_Edge, Atpd_data_q );
        InvPath ( Data_ISchd, Data_Edge, Atpd_data_q );

        BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

        -- ------------------------------------
        -- Compute function and propation delaq
        -- ------------------------------------
        NewValue  := VitalDECODER4 ( Data, Enable, ResultMap );
        new_schd := VitalDECODER4 ( Data_BSchd, Data_ISchd, Enable_Schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
      END IF;
    END;
--
    PROCEDURE VitalDECODER8  (
            SIGNAL              q : OUT std_logic_vector8;
            SIGNAL           Data :  IN std_logic_vector3;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE NewValue        : std_logic_vector8;
        VARIABLE Glitch_Data    : GlitchArray8;
        VARIABLE new_schd       : SchedArray8;
        VARIABLE Dly, Glch      : TimeArray8;
        VARIABLE Enable_Schd    : SchedType;
        VARIABLE Enable_Edge    : EdgeType;
        VARIABLE Data_Edge      : EdgeArray3;
```

 Copyright © 2001 IEEE. All rights reserved.

```
            VARIABLE Data_BSchd, Data_ISchd : SchedArray3;
            ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
            VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
        BEGIN
          -- ------------------------------------------------------------------------
          --  Check if ALL zero delay paths, use simple model
          --    ( No delay selection, glitch detection required )
          -- ------------------------------------------------------------------------
          IF (tpd_enable_q /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
          ELSE
              FOR i IN Data'RANGE LOOP
              IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                  AllZeroDelay := FALSE;
                  EXIT;
              END IF;
              END LOOP;
          END IF;
          IF (AllZeroDelay) THEN LOOP
              q <= VitalDECODER(Data, Enable, ResultMap);
              WAIT ON Data, Enable;
          END LOOP;
          ELSE

            -- --------------------------------------
            -- Initialize delay schedules
            -- --------------------------------------
            FOR n IN Data'RANGE LOOP
                BufPath ( Data_BSchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
                InvPath ( Data_ISchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
            END LOOP;
            BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

          LOOP
            -- --------------------------------------
            -- Process input signals
            --   get edge values
            --   re-evaluate output schedules
            -- --------------------------------------
            GetEdge ( Data, LastData, Data_Edge );
            BufPath ( Data_BSchd, Data_Edge, Atpd_data_q );
            InvPath ( Data_ISchd, Data_Edge, Atpd_data_q );

            BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

            -- ------------------------------------
            -- Compute function and propation delaq
            -- ------------------------------------
            NewValue  := VitalDECODER8 ( Data, Enable, ResultMap );
            new_schd := VitalDECODER8 ( Data_BSchd, Data_ISchd, Enable_Schd );

            -- ------------------------------------------------------
            -- Assign Outputs
            --  get delays to new value and possable glitch
            --  schedule output change with On Event glitch detection
            -- ------------------------------------------------------
            GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
            VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                                 PrimGlitchMode, GlitchDelay=>Glch );

            WAIT ON Data, Enable;
          END LOOP;
          END IF; --SN
        END;
    --
        PROCEDURE VitalDECODER   (
                SIGNAL              q : OUT std_logic_vector;
                SIGNAL           Data :  IN std_logic_vector;
                SIGNAL         Enable :  IN std_ulogic;
                CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
                CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT    ResultMap :  IN VitalResultMapType
                                            := VitalDefaultResultMap
        ) IS
            VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
            VARIABLE NewValue        : std_logic_vector(q'RANGE);
            VARIABLE Glitch_Data    : GlitchDataArrayType(q'RANGE);
            VARIABLE new_schd       : SchedArray(q'RANGE);
            VARIABLE Dly, Glch      : VitalTimeArray(q'RANGE);
            VARIABLE Enable_Schd    : SchedType;
            VARIABLE Enable_Edge    : EdgeType;
```

Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE Data_Edge      : EdgeArray(Data'RANGE);
    VARIABLE Data_BSchd, Data_ISchd : SchedArray(Data'RANGE);
    ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
    VARIABLE AllZeroDelay  : BOOLEAN := TRUE;
BEGIN
  -- ------------------------------------------------------------------------
  --  Check if ALL zero delay paths, use simple model
  --   ( No delay selection, glitch detection required )
  -- ------------------------------------------------------------------------
  IF (tpd_enable_q /= VitalZeroDelay01) THEN
      AllZeroDelay := FALSE;
  ELSE
      FOR i IN Data'RANGE LOOP
      IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
          AllZeroDelay := FALSE;
          EXIT;
      END IF;
      END LOOP;
  END IF;
  IF (AllZeroDelay) THEN LOOP
      q <= VitalDECODER(Data, Enable, ResultMap);
      WAIT ON Data, Enable;
  END LOOP;
  ELSE
    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    FOR n IN Data'RANGE LOOP
        BufPath ( Data_BSchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        InvPath ( Data_ISchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
    END LOOP;
    BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    GetEdge ( Data, LastData, Data_Edge );
    BufPath ( Data_BSchd, Data_Edge, Atpd_data_q );
    InvPath ( Data_ISchd, Data_Edge, Atpd_data_q );

    BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

    -- ------------------------------------
    -- Compute function and propation delaq
    -- ------------------------------------
    NewValue  := VitalDECODER ( Data, Enable, ResultMap );
    new_schd := VitalDECODER ( Data_BSchd, Data_ISchd, Enable_Schd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON Data, Enable;
  END LOOP;
  END IF;
END;

-- ------------------------------------------------------------------------
FUNCTION VitalTruthTable  (
        CONSTANT TruthTable   : IN VitalTruthTableType;
        CONSTANT DataIn       : IN std_logic_vector
      ) RETURN std_logic_vector IS

    CONSTANT InputSize   : INTEGER := DataIn'LENGTH;
    CONSTANT OutSize     : INTEGER := TruthTable'LENGTH(2) - InputSize;
    VARIABLE ReturnValue : std_logic_vector(OutSize - 1 DOWNTO 0)
                         := (OTHERS => 'X');
    VARIABLE DataInAlias : std_logic_vector(0 TO InputSize - 1)
                         :=  To_X01(DataIn);
    VARIABLE Index       : INTEGER;
    VARIABLE Err         : BOOLEAN := FALSE;

    -- This needs to be done since the TableLookup arrays must be
```

 Copyright © 2001 IEEE. All rights reserved.

```
    -- ascending starting with 0
    VARIABLE TableAlias  : VitalTruthTableType(0 TO (TruthTable'LENGTH(1)-1),
                                                0 TO (TruthTable'LENGTH(2)-1))
                        := TruthTable;

BEGIN
    -- search through each row of the truth table
    IF OutSize > 0 THEN
      ColLoop:
        FOR i IN TableAlias'RANGE(1) LOOP

        RowLoop: -- Check each input element of the entry
          FOR j IN 0 TO InputSize LOOP

            IF (j = InputSize) THEN -- This entry matches
                -- Return the Result
                Index := 0;
                FOR k IN TruthTable'LENGTH(2) - 1 DOWNTO InputSize LOOP
                    TruthOutputX01Z ( TableAlias(i,k),
                                      ReturnValue(Index), Err);
                    EXIT WHEN Err;
                    Index := Index + 1;
                END LOOP;

                IF Err THEN
                    ReturnValue := (OTHERS => 'X');
                END IF;
                RETURN ReturnValue;
            END IF;
            IF NOT ValidTruthTableInput(TableAlias(i,j)) THEN
                VitalError ( "VitalTruthTable", ErrInpSym,
                             To_TruthChar(TableAlias(i,j)) );
                EXIT ColLoop;
            END IF;
            EXIT RowLoop WHEN NOT ( TruthTableMatch( DataInAlias(j),
                                                     TableAlias(i, j)));
          END LOOP RowLoop;
        END LOOP ColLoop;

    ELSE
        VitalError ( "VitalTruthTable", ErrTabWidSml );
    END IF;
    RETURN ReturnValue;
END VitalTruthTable;

FUNCTION VitalTruthTable  (
        CONSTANT TruthTable   : IN VitalTruthTableType;
        CONSTANT DataIn       : IN std_logic_vector
      ) RETURN std_logic IS

    CONSTANT InputSize  : INTEGER := DataIn'LENGTH;
    CONSTANT OutSize    : INTEGER := TruthTable'LENGTH(2) - InputSize;
    VARIABLE TempResult : std_logic_vector(OutSize - 1 DOWNTO 0)
                        := (OTHERS => 'X');
BEGIN
    IF (OutSize > 0) THEN
        TempResult := VitalTruthTable(TruthTable, DataIn);
        IF ( 1 > OutSize) THEN
            VitalError ( "VitalTruthTable", ErrTabResSml );
        ELSIF ( 1 < OutSize) THEN
            VitalError ( "VitalTruthTable", ErrTabResLrg );
        END IF;
        RETURN (TempResult(0));
    ELSE
        VitalError ( "VitalTruthTable", ErrTabWidSml );
        RETURN 'X';
    END IF;
END VitalTruthTable;

PROCEDURE VitalTruthTable (
        SIGNAL   Result     : OUT std_logic_vector;
        CONSTANT TruthTable : IN  VitalTruthTableType;
        SIGNAL   DataIn     : IN  std_logic_vector        -- IR#236
) IS
    CONSTANT ResLeng     : INTEGER := Result'LENGTH;
    CONSTANT ActResLen   : INTEGER := TruthTable'LENGTH(2) - DataIn'LENGTH;
    CONSTANT FinalResLen : INTEGER := Minimum(ActResLen, ResLeng);
    VARIABLE TempResult  : std_logic_vector(ActResLen - 1 DOWNTO 0)
                         := (OTHERS => 'X');

BEGIN
```

Copyright © 2001 IEEE. All rights reserved.

```
        TempResult := VitalTruthTable(TruthTable, DataIn);

        IF (ResLeng > ActResLen) THEN
            VitalError ( "VitalTruthTable", ErrTabResSml );
        ELSIF (ResLeng < ActResLen) THEN
            VitalError ( "VitalTruthTable", ErrTabResLrg );
        END IF;
        TempResult(FinalResLen-1 DOWNTO 0) := TempResult(FinalResLen-1 DOWNTO 0);
        Result <= TempResult;

    END VitalTruthTable;

    PROCEDURE VitalTruthTable (
            SIGNAL   Result     : OUT std_logic;
            CONSTANT TruthTable : IN VitalTruthTableType;
            SIGNAL DataIn       : IN std_logic_vector        -- IR#236
    ) IS

        CONSTANT ActResLen  : INTEGER := TruthTable'LENGTH(2) - DataIn'LENGTH;
        VARIABLE TempResult : std_logic_vector(ActResLen - 1 DOWNTO 0)
                              := (OTHERS => 'X');

    BEGIN
        TempResult := VitalTruthTable(TruthTable, DataIn);

        IF ( 1 > ActResLen) THEN
            VitalError ( "VitalTruthTable", ErrTabResSml );
        ELSIF ( 1 < ActResLen) THEN
            VitalError ( "VitalTruthTable", ErrTabResLrg );
        END IF;
        IF (ActResLen >  0) THEN
            Result <= TempResult(0);
        END IF;

    END VitalTruthTable;

    -- ------------------------------------------------------------------------
    PROCEDURE VitalStateTable (
            VARIABLE Result         : INOUT std_logic_vector;
            VARIABLE PreviousDataIn : INOUT std_logic_vector;
            CONSTANT StateTable     : IN VitalStateTableType;
            CONSTANT DataIn         : IN std_logic_vector;
            CONSTANT NumStates      : IN NATURAL
    ) IS

        CONSTANT InputSize     : INTEGER := DataIn'LENGTH;
        CONSTANT OutSize       : INTEGER
                                 := StateTable'LENGTH(2) - InputSize - NumStates;
        CONSTANT ResLeng       : INTEGER := Result'LENGTH;
        VARIABLE DataInAlias   : std_logic_vector(0 TO DataIn'LENGTH-1)
                                 := To_X01(DataIn);
        VARIABLE PrevDataAlias : std_logic_vector(0 TO PreviousDataIn'LENGTH-1)
                                 := To_X01(PreviousDataIn);
        VARIABLE ResultAlias   : std_logic_vector(0 TO ResLeng-1)
                                 := To_X01(Result);
        VARIABLE ExpResult     : std_logic_vector(0 TO OutSize-1);

    BEGIN
        IF (PreviousDataIn'LENGTH < DataIn'LENGTH) THEN
            VitalError ( "VitalStateTable", ErrVctLng, "PreviousDataIn<DataIn");

            ResultAlias := (OTHERS => 'X');
            Result := ResultAlias;

        ELSIF (OutSize <= 0) THEN
            VitalError ( "VitalStateTable", ErrTabWidSml );

            ResultAlias := (OTHERS => 'X');
            Result := ResultAlias;

        ELSE
            IF (ResLeng > OutSize) THEN
                VitalError ( "VitalStateTable", ErrTabResSml );
            ELSIF (ResLeng < OutSize) THEN
                VitalError ( "VitalStateTable", ErrTabResLrg );
            END IF;

            ExpResult := StateTableLookUp ( StateTable, DataInAlias,
                                            PrevDataAlias, NumStates,
                                            ResultAlias);
            ResultAlias := (OTHERS => 'X');
```

 Copyright © 2001 IEEE. All rights reserved.

```
        ResultAlias ( Maximum(0, ResLeng - OutSize) TO ResLeng - 1)
               := ExpResult(Maximum(0, OutSize - ResLeng) TO OutSize-1);

        Result := ResultAlias;
        PrevDataAlias(0 TO InputSize - 1) := DataInAlias;
        PreviousDataIn := PrevDataAlias;

    END IF;
END VitalStateTable;


PROCEDURE VitalStateTable (
        VARIABLE Result         : INOUT std_logic;        -- states
     VARIABLE PreviousDataIn : INOUT std_logic_vector; -- previous inputs and states
       CONSTANT StateTable     : IN VitalStateTableType; -- User's StateTable data
        CONSTANT DataIn         : IN std_logic_vector     -- Inputs
) IS

    VARIABLE ResultAlias : std_logic_vector(0 TO 0);
BEGIN
    ResultAlias(0) := Result;
    VitalStateTable ( StateTable     => StateTable,
                      DataIn         => DataIn,
                      NumStates      => 1,
                      Result         => ResultAlias,
                      PreviousDataIn => PreviousDataIn
                    );
    Result := ResultAlias(0);

END VitalStateTable;

PROCEDURE VitalStateTable (
        SIGNAL   Result     : INOUT std_logic_vector;
        CONSTANT StateTable : IN VitalStateTableType;
        SIGNAL   DataIn     : IN std_logic_vector;
        CONSTANT NumStates  : IN NATURAL
) IS

    CONSTANT InputSize   : INTEGER := DataIn'LENGTH;
    CONSTANT OutSize     : INTEGER
                           := StateTable'LENGTH(2) - InputSize - NumStates;
    CONSTANT ResLeng     : INTEGER := Result'LENGTH;

    VARIABLE PrevData    : std_logic_vector(0 TO DataIn'LENGTH-1)
                           := (OTHERS => 'X');
    VARIABLE DataInAlias : std_logic_vector(0 TO DataIn'LENGTH-1);
    VARIABLE ResultAlias : std_logic_vector(0 TO ResLeng-1);
    VARIABLE ExpResult   : std_logic_vector(0 TO OutSize-1);

BEGIN
    IF (OutSize <= 0) THEN
        VitalError ( "VitalStateTable", ErrTabWidSml );

        ResultAlias  := (OTHERS => 'X');
        Result <= ResultAlias;

    ELSE
        IF (ResLeng > OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResSml );
        ELSIF (ResLeng < OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResLrg );
        END IF;

        LOOP
            DataInAlias := To_X01(DataIn);
            ResultAlias := To_X01(Result);
            ExpResult   := StateTableLookUp ( StateTable, DataInAlias,
                                              PrevData, NumStates,
                                              ResultAlias);
            ResultAlias := (OTHERS => 'X');
            ResultAlias(Maximum(0, ResLeng - OutSize) TO ResLeng-1)
                   := ExpResult(Maximum(0, OutSize - ResLeng) TO OutSize-1);

            Result   <= ResultAlias;
            PrevData := DataInAlias;

            WAIT ON DataIn;
        END LOOP;

    END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
    END VitalStateTable;

    PROCEDURE VitalStateTable (
            SIGNAL   Result     : INOUT std_logic;
            CONSTANT StateTable : IN VitalStateTableType;
            SIGNAL   DataIn     : IN std_logic_vector
    ) IS

        CONSTANT InputSize   : INTEGER := DataIn'LENGTH;
        CONSTANT OutSize     : INTEGER := StateTable'LENGTH(2) - InputSize-1;

        VARIABLE PrevData    : std_logic_vector(0 TO DataIn'LENGTH-1)
                             := (OTHERS => 'X');
        VARIABLE DataInAlias : std_logic_vector(0 TO DataIn'LENGTH-1);
        VARIABLE ResultAlias : std_logic_vector(0 TO 0);
        VARIABLE ExpResult   : std_logic_vector(0 TO OutSize-1);

    BEGIN
        IF (OutSize <= 0) THEN
            VitalError ( "VitalStateTable", ErrTabWidSml );

            Result <= 'X';

        ELSE
            IF ( 1 > OutSize) THEN
                VitalError ( "VitalStateTable", ErrTabResSml );
            ELSIF ( 1 < OutSize) THEN
                VitalError ( "VitalStateTable", ErrTabResLrg );
            END IF;

            LOOP
                ResultAlias(0) := To_X01(Result);
                DataInAlias := To_X01(DataIn);
                ExpResult   := StateTableLookUp ( StateTable, DataInAlias,
                                                  PrevData, 1, ResultAlias);

                Result   <= ExpResult(OutSize-1);
                PrevData := DataInAlias;

                WAIT ON DataIn;
            END LOOP;
        END IF;

    END VitalStateTable;

    -- ------------------------------------------------------------------------
    -- std_logic resolution primitive
    -- ------------------------------------------------------------------------
    PROCEDURE VitalResolve (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector  --IR236 4/2/98
    ) IS
        VARIABLE uData : std_ulogic_vector(Data'RANGE);
    BEGIN
        FOR i IN Data'RANGE LOOP
            uData(i) := Data(i);
        END LOOP;
        q <= resolved(uData);
    END;

END VITAL_Primitives;
```

13.4 VITAL_Primitives package body

```
-------------------------------------------------------------------------------
-- Title        : Standard VITAL_Primitives Package
--              : $Revision: 1.2 $
--              :
-- Library      : VITAL
--              :
-- Developers   : IEEE DASC Timing Working Group (TWG), PAR 1076.4
--              :
-- Purpose      : This packages defines standard types, constants, functions
--              : and procedures for use in developing ASIC models.
--              : Specifically a set of logic primitives are defined.
--              :
-- ----------------------------------------------------------------------------
```

 Copyright © 2001 IEEE. All rights reserved.

```
--
-- -----------------------------------------------------------------------------
-- Modification History :
-- -----------------------------------------------------------------------------
-- Version No:|Auth:| Mod.Date:| Changes Made:
--   v95.0 A  |     | 06/02/95 | Initial ballot draft 1995
--   v95.1    |     | 08/31/95 | #204 - glitch detection prior to OutputMap
-- -----------------------------------------------------------------------------
--   v95.2    | ddl | 09/14/96 | #223 - single input prmtvs use on-detect
--            |     |          |        instead of glitch-on-event behavior
--   v95.3    | ddl | 09/24/96 | #236 - VitalTruthTable DataIn should be of
--            |     |          |        of class SIGNAL
--   v95.4    | ddl | 01/16/97 | #243 - index constraint error in nbit xor/xnor
--   v99.1    | dbb | 03/31/99 | Updated for VHDL 93
-- -----------------------------------------------------------------------------

LIBRARY STD;
USE STD.TEXTIO.ALL;

PACKAGE BODY VITAL_Primitives IS
    -- ------------------------------------------------------------------------
    --  Default values for Primitives
    -- ------------------------------------------------------------------------
    --  default values for delay parameters
    CONSTANT VitalDefDelay01  : VitalDelayType01  := VitalZeroDelay01;
    CONSTANT VitalDefDelay01Z : VitalDelayType01Z := VitalZeroDelay01Z;

    TYPE VitalTimeArray IS ARRAY (NATURAL RANGE <>) OF TIME;

    --  default primitive model operation parameters
    --  Glitch detection/reporting
    TYPE VitalGlitchModeType IS ( MessagePlusX, MessageOnly, XOnly, NoGlitch);
    CONSTANT PrimGlitchMode : VitalGlitchModeType   := XOnly;

    -- ------------------------------------------------------------------------
    -- Local Type and Subtype Declarations
    -- ------------------------------------------------------------------------
    ---------------------------------------------------------------------------
    -- enumeration value representing the transition or level of the signal.
    --  See function 'GetEdge'
    ---------------------------------------------------------------------------
    TYPE EdgeType IS ( 'U',   -- Uninitialized level
                       'X',   -- Unknown level
                       '0',   -- low level
                       '1',   -- high level
                       '\',   -- 1 to 0 falling edge
                       '/',   -- 0 to 1 rising  edge
                       'F',   -- * to 0 falling edge
                       'R',   -- * to 1 rising  edge
                       'f',   -- rising  to X edge
                       'r',   -- falling to X edge
                       'x',   -- Unknown edge (ie U->X)
                       'V'    -- Timing violation edge
                     );
    TYPE EdgeArray  IS ARRAY ( NATURAL RANGE <> ) OF EdgeType;

    TYPE EdgeX1Table IS ARRAY ( EdgeType                            ) OF EdgeType;
    TYPE EdgeX2Table IS ARRAY ( EdgeType, EdgeType                  ) OF EdgeType;
    TYPE EdgeX3Table IS ARRAY ( EdgeType, EdgeType, EdgeType        ) OF EdgeType;
    TYPE EdgeX4Table IS ARRAY (EdgeType,EdgeType,EdgeType,EdgeType) OF EdgeType;

    TYPE LogicToEdgeT  IS ARRAY(std_ulogic, std_ulogic) OF EdgeType;
    TYPE LogicToLevelT IS ARRAY(std_ulogic ) OF EdgeType;

    TYPE GlitchDataType IS
      RECORD
        SchedTime    : TIME;
        GlitchTime   : TIME;
        SchedValue   : std_ulogic;
        CurrentValue    : std_ulogic;
      END RECORD;
    TYPE GlitchDataArrayType IS ARRAY (NATURAL RANGE <>)
         OF GlitchDataType;

    -- Enumerated type used in selection of output path delays
    TYPE SchedType  IS
      RECORD
        inp0  : TIME;   -- time (abs) of output change due to input change to 0
        inp1  : TIME;   -- time (abs) of output change due to input change to 1
        InpX  : TIME;   -- time (abs) of output change due to input change to X
        Glch0 : TIME;   -- time (abs) of output glitch due to input change to 0
```

Copyright © 2001 IEEE. All rights reserved.

```
      Glch1 : TIME;   -- time (abs) of output glitch due to input change to 0
    END RECORD;

  TYPE SchedArray  IS ARRAY ( NATURAL RANGE <> ) OF SchedType;
  CONSTANT DefSchedType : SchedType := (TIME'HIGH, TIME'HIGH, 0 ns,0 ns,0 ns);
  CONSTANT DefSchedAnd  : SchedType := (TIME'HIGH, 0 ns,0 ns, TIME'HIGH,0 ns);

  -- Constrained array declarations (common sizes used by primitives)
  SUBTYPE SchedArray2 IS SchedArray(1 DOWNTO 0);
  SUBTYPE SchedArray3 IS SchedArray(2 DOWNTO 0);
  SUBTYPE SchedArray4 IS SchedArray(3 DOWNTO 0);
  SUBTYPE SchedArray8 IS SchedArray(7 DOWNTO 0);

  SUBTYPE TimeArray2 IS VitalTimeArray(1 DOWNTO 0);
  SUBTYPE TimeArray3 IS VitalTimeArray(2 DOWNTO 0);
  SUBTYPE TimeArray4 IS VitalTimeArray(3 DOWNTO 0);
  SUBTYPE TimeArray8 IS VitalTimeArray(7 DOWNTO 0);

  SUBTYPE GlitchArray2 IS GlitchDataArrayType(1 DOWNTO 0);
  SUBTYPE GlitchArray3 IS GlitchDataArrayType(2 DOWNTO 0);
  SUBTYPE GlitchArray4 IS GlitchDataArrayType(3 DOWNTO 0);
  SUBTYPE GlitchArray8 IS GlitchDataArrayType(7 DOWNTO 0);

  SUBTYPE EdgeArray2 IS EdgeArray(1 DOWNTO 0);
  SUBTYPE EdgeArray3 IS EdgeArray(2 DOWNTO 0);
  SUBTYPE EdgeArray4 IS EdgeArray(3 DOWNTO 0);
  SUBTYPE EdgeArray8 IS EdgeArray(7 DOWNTO 0);

  CONSTANT DefSchedArray2 : SchedArray2 :=
                           (OTHERS=> (0 ns, 0 ns, 0 ns, 0 ns, 0 ns));

  TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

  CONSTANT InitialEdge : LogicToLevelT := (
          '1'|'H'  => 'R',
          '0'|'L'  => 'F',
          OTHERS   => 'x'
   );

  CONSTANT LogicToEdge  : LogicToEdgeT  := (  -- previous, current
  --  old \ new: U    X    0    1    Z    W    L    H    -
      'U'  =>  ( 'U', 'x', 'F', 'R', 'x', 'x', 'F', 'R', 'x' ),
      'X'  =>  ( 'x', 'X', 'F', 'R', 'x', 'X', 'F', 'R', 'X' ),
      '0'  =>  ( 'r', 'r', '0', '/', 'r', 'r', '0', '/', 'r' ),
      '1'  =>  ( 'f', 'f', '\', '1', 'f', 'f', '\', '1', 'f' ),
      'Z'  =>  ( 'x', 'X', 'F', 'R', 'X', 'x', 'F', 'R', 'x' ),
      'W'  =>  ( 'x', 'X', 'F', 'R', 'x', 'X', 'F', 'R', 'X' ),
      'L'  =>  ( 'r', 'r', '0', '/', 'r', 'r', '0', '/', 'r' ),
      'H'  =>  ( 'f', 'f', '\', '1', 'f', 'f', '\', '1', 'f' ),
      '-'  =>  ( 'x', 'X', 'F', 'R', 'x', 'X', 'F', 'R', 'X' )
  );
  CONSTANT LogicToLevel : LogicToLevelT := (
          '1'|'H'  => '1',
          '0'|'L'  => '0',
          'U'      => 'U',
          OTHERS   => 'X'
   );

  -- ------------------------------------
  -- 3-state logic tables
  -- ------------------------------------
  CONSTANT BufIf0_Table : stdlogic_table :=
      -- enable        data       value
      ( '1'|'H'    => ( OTHERS  => 'Z' ),
        '0'|'L'    => ( '1'|'H'  => '1',
                        '0'|'L'  => '0',
                        'U'      => 'U',
                        OTHERS   => 'X' ),
        'U'        => ( OTHERS  => 'U' ),
        OTHERS     => ( OTHERS  => 'X' ) );
  CONSTANT BufIf1_Table : stdlogic_table :=
      -- enable        data       value
      ( '0'|'L'    => ( OTHERS  => 'Z' ),
        '1'|'H'    => ( '1'|'H'  => '1',
                        '0'|'L'  => '0',
                        'U'      => 'U',
                        OTHERS   => 'X' ),
        'U'        => ( OTHERS  => 'U' ),
        OTHERS     => ( OTHERS  => 'X' ) );
  CONSTANT InvIf0_Table : stdlogic_table :=
      -- enable        data       value
```

 Copyright © 2001 IEEE. All rights reserved.

```
    ( ‘1’ |’ H’    => ( OTHERS  => ‘Z’ ),
      ‘0’ |’ L’    => ( ‘1’ |’ H’ => ‘0’ ,
                        ‘0’ |’ L’ => ‘1’ ,
                        ‘U’       => ‘U’ ,
                        OTHERS  => ‘X’ ),
      ‘U’          => ( OTHERS  => ‘U’ ),
      OTHERS       => ( OTHERS  => ‘X’ ) );
CONSTANT InvIf1_Table : stdlogic_table :=
    -- enable        data       value
    ( ‘0’ |’ L’    => ( OTHERS  => ‘Z’ ),
      ‘1’ |’ H’    => ( ‘1’ |’ H’ => ‘0’ ,
                        ‘0’ |’ L’ => ‘1’ ,
                        ‘U’       => ‘U’ ,
                        OTHERS  => ‘X’ ),
      ‘U’          => ( OTHERS  => ‘U’ ),
      OTHERS       => ( OTHERS  => ‘X’ ) );


TYPE To_StateCharType IS ARRAY (VitalStateSymbolType) OF CHARACTER;
CONSTANT To_StateChar : To_StateCharType :=
 ( ‘/’ , ‘\’ , ‘P’ , ‘N’ , ‘r’ , ‘f’ , ‘p’ , ‘n’ , ‘R’ , ‘F’ , ‘^’ , ‘v’ ,
   ‘E’ , ‘A’ , ‘D’ , ‘*’ , ‘X’ , ‘0’ , ‘1’ , ‘-’ , ‘B’ , ‘Z’ , ‘S’ );
TYPE To_TruthCharType IS ARRAY (VitalTruthSymbolType) OF CHARACTER;
CONSTANT To_TruthChar : To_TruthCharType :=
 ( ‘X’ , ‘0’ , ‘1’ , ‘-’ , ‘B’ , ‘Z’ );

TYPE TruthTableOutMapType IS ARRAY (VitalTruthSymbolType) OF std_ulogic;
CONSTANT TruthTableOutMap : TruthTableOutMapType :=
   --  ‘X’ , ‘0’ , ‘1’ , ‘-’ , ‘B’ , ‘Z’
     ( ‘X’ , ‘0’ , ‘1’ , ‘X’ , ‘-’ , ‘Z’ );

TYPE StateTableOutMapType IS ARRAY (VitalStateSymbolType) OF std_ulogic;
-- does conversion to X01Z or ‘-’ if invalid
CONSTANT StateTableOutMap : StateTableOutMapType :=
 -- ‘/’  ‘\’  ‘P’  ‘N’  ‘r’  ‘f’  ‘p’  ‘n’  ‘R’  ‘F’  ‘^’  ‘v’
 -- ‘E’  ‘A’  ‘D’  ‘*’  ‘X’  ‘0’  ‘1’  ‘-’  ‘B’  ‘Z’  ‘S’
  ( ‘-’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,’ -’ ,
    ‘-’ ,’ -’ ,’ -’ ,’ -’ ,’ X’ ,’ 0’ ,’ 1’ ,’ X’ ,’ -’ ,’ Z’ ,’ W’ );

-- ------------------------------------------------------------------------
TYPE ValidTruthTableInputType IS ARRAY (VitalTruthSymbolType) OF BOOLEAN;
-- checks if a symbol IS valid for the stimulus portion of a truth table
CONSTANT ValidTruthTableInput : ValidTruthTableInputType :=
   -- ‘X’     ‘0’     ‘1’     ‘-’     ‘B’     ‘Z’
   (  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  FALSE );

TYPE TruthTableMatchType IS ARRAY (X01, VitalTruthSymbolType) OF BOOLEAN;
-- checks if an input matches th corresponding truth table symbol
-- use: TruthTableMatch(input_converted_to_X01, truth_table_stimulus_symbol)
CONSTANT TruthTableMatch : TruthTableMatchType  :=  (
   -- X,     0,     1,     -      B      Z
   (  TRUE,  FALSE, FALSE, TRUE,  FALSE, FALSE  ),  -- X
   (  FALSE, TRUE,  FALSE, TRUE,  TRUE,  FALSE  ),  -- 0
   (  FALSE, FALSE, TRUE,  TRUE,  TRUE,  FALSE  )   -- 1
);

-- ------------------------------------------------------------------------
TYPE ValidStateTableInputType IS ARRAY (VitalStateSymbolType) OF BOOLEAN;
CONSTANT ValidStateTableInput : ValidStateTableInputType :=
   -- ‘/’ ,   ‘\’ ,   ‘P’ ,   ‘N’ ,   ‘r’ ,   ‘f’ ,
  (   TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,
   -- ‘p’ ,   ‘n’ ,   ‘R’ ,   ‘F’ ,   ‘^’ ,   ‘v’ ,
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,
   -- ‘E’ ,   ‘A’ ,    ‘D’ ,  ‘*’ ,
      TRUE,  TRUE,  TRUE,  TRUE,
   -- ‘X’ ,   ‘0’ ,   ‘1’ ,   ‘-’ ,   ‘B’ ,   ‘Z’ ,
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE, FALSE,
   -- ‘S’
      TRUE );

CONSTANT ValidStateTableState : ValidStateTableInputType :=
   -- ‘/’ ,   ‘\’ ,   ‘P’ ,   ‘N’ ,   ‘r’ ,   ‘f’ ,
  (   FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
   -- ‘p’ ,   ‘n’ ,   ‘R’ ,   ‘F’ ,   ‘^’ ,   ‘v’ ,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
   -- ‘E’ ,   ‘A’ ,    ‘D’ ,  ‘*’ ,
      FALSE, FALSE, FALSE, FALSE,
   -- ‘X’ ,   ‘0’ ,   ‘1’ ,   ‘-’ ,   ‘B’ ,   ‘Z’ ,
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  FALSE,
   -- ‘S’
      FALSE );
```

Copyright © 2001 IEEE. All rights reserved.

```
TYPE StateTableMatchType IS ARRAY (X01,X01,VitalStateSymbolType) OF BOOLEAN;
-- last value, present value, table symbol
CONSTANT StateTableMatch : StateTableMatchType :=  (
  ( -- X (lastvalue)
 -- /     \     P     N     r     f
 -- p     n     R     F     ^     v
 -- E     A     D     *
 -- X     0     1     -     B     Z     S
  (FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,
   TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE),
  (FALSE,FALSE,FALSE,TRUE, FALSE,FALSE,
   FALSE,FALSE,FALSE,TRUE, FALSE,TRUE,
   TRUE, FALSE,TRUE, TRUE,
   FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,FALSE),
  (FALSE,FALSE,TRUE, FALSE,FALSE,FALSE,
   FALSE,FALSE,TRUE, FALSE,TRUE, FALSE,
   TRUE, TRUE, FALSE,TRUE,
   FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,FALSE)
  ),

  (-- 0 (lastvalue)
 -- /     \     P     N     r     f
 -- p     n     R     F     ^     v
 -- E     A     D     *
 -- X     0     1     -     B     Z     S
  (FALSE,FALSE,FALSE,FALSE,TRUE, FALSE,
   TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
   FALSE,TRUE, FALSE,TRUE,
   TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE),
  (FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,
   FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,TRUE ),
  (TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
   TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,TRUE,
   FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,FALSE)
  ),

  (-- 1 (lastvalue)
 -- /     \     P     N     r     f
 -- p     n     R     F     ^     v
 -- E     A     D     *
 -- X     0     1     -     B     Z     S
  (FALSE,FALSE,FALSE,FALSE,FALSE,TRUE ,
   FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
   FALSE,FALSE,TRUE, TRUE,
   TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE),
  (FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
   FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
   FALSE,FALSE,FALSE,TRUE,
   FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,FALSE),
  (FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,FALSE,FALSE,
   FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,TRUE )
  )
  );

TYPE Logic_UX01Z_Table IS ARRAY (std_ulogic) OF UX01Z;
----------------------------------------------------------
-- table name : cvt_to_x01z
-- parameters :  std_ulogic  -- some logic value
-- returns    :  UX01Z       -- state value of logic value
-- purpose    :  to convert state-strength to state only
----------------------------------------------------------
CONSTANT cvt_to_ux01z : Logic_UX01Z_Table :=
                          ('U','X','0','1','Z','X','0','1','X' );

TYPE LogicCvtTableType IS ARRAY (std_ulogic) OF CHARACTER;
CONSTANT LogicCvtTable : LogicCvtTableType
                 := ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

---------------------------------------------------------------------
-- LOCAL Utilities
---------------------------------------------------------------------
-- ---------------------------------------------------------------------
--  FUNCTION  NAME  :  MINIMUM
```

 Copyright © 2001 IEEE. All rights reserved.

```
--
--  PARAMETERS      :  in1, in2  - integer, time
--
--  DESCRIPTION     :  return smaller of in1 and in2
-- ------------------------------------------------------------------------
FUNCTION Minimum (
        CONSTANT in1, in2 : INTEGER
      ) RETURN INTEGER IS
BEGIN
   IF (in1 < in2) THEN
      RETURN in1;
   END IF;
   RETURN in2;
END;
-- ------------------------------------------------------------------------
FUNCTION Minimum (
        CONSTANT t1,t2 : IN TIME
      ) RETURN TIME IS
BEGIN
    IF ( t1 < t2 ) THEN RETURN (t1); ELSE RETURN (t2); END IF;
END Minimum;

-- ------------------------------------------------------------------------
--  FUNCTION  NAME  :  MAXIMUM
--
--  PARAMETERS      :  in1, in2  - integer, time
--
--  DESCRIPTION     :  return larger of in1 and in2
-- ------------------------------------------------------------------------
FUNCTION Maximum (
        CONSTANT in1, in2 : INTEGER
      ) RETURN INTEGER IS
BEGIN
   IF (in1 > in2) THEN
      RETURN in1;
   END IF;
   RETURN in2;
END;
-----------------------------------------------------------------------
FUNCTION Maximum (
        CONSTANT t1,t2 : IN TIME
      ) RETURN TIME IS
BEGIN
    IF ( t1 > t2 ) THEN RETURN (t1); ELSE RETURN (t2); END IF;
END Maximum;

-----------------------------------------------------------------------
IMPURE FUNCTION GlitchMinTime (
        CONSTANT Time1, Time2 : IN TIME
      ) RETURN TIME IS
BEGIN
    IF ( Time1 >= NOW ) THEN
            IF ( Time2 >= NOW ) THEN
              RETURN Minimum ( Time1, Time2);
            ELSE
              RETURN Time1;
            END IF;
    ELSE
            IF ( Time2 >= NOW ) THEN
               RETURN Time2;
            ELSE
               RETURN 0 ns;
            END IF;
    END IF;
END;

--------------------------------------------------------------------
-- Error Message Types and Tables
--------------------------------------------------------------------
TYPE VitalErrorType IS (
    ErrNegDel,
    ErrInpSym,
    ErrOutSym,
    ErrStaSym,
    ErrVctLng,
    ErrTabWidSml,
    ErrTabWidLrg,
    ErrTabResSml,
    ErrTabResLrg
);
```

Copyright © 2001 IEEE. All rights reserved.

```
TYPE VitalErrorSeverityType IS ARRAY (VitalErrorType) OF SEVERITY_LEVEL;
CONSTANT VitalErrorSeverity : VitalErrorSeverityType := (
    ErrNegDel    => WARNING,
    ErrInpSym    => ERROR,
    ErrOutSym    => ERROR,
    ErrStaSym    => ERROR,
    ErrVctLng    => ERROR,
    ErrTabWidSml => ERROR,
    ErrTabWidLrg => WARNING,
    ErrTabResSml => WARNING,
    ErrTabResLrg => WARNING
);

CONSTANT MsgNegDel : STRING :=
  "Negative delay. New output value not scheduled. Output signal is: ";
CONSTANT MsgInpSym : STRING :=
  "Illegal symbol in the input portion of a Truth/State table.";
CONSTANT MsgOutSym : STRING :=
  "Illegal symbol in the output portion of a Truth/State table.";
CONSTANT MsgStaSym : STRING :=
  "Illegal symbol in the state portion of a State table.";
CONSTANT MsgVctLng : STRING :=
  "Vector (array) lengths not equal. ";
CONSTANT MsgTabWidSml : STRING :=
  "Width of the Truth/State table is too small.";
CONSTANT MsgTabWidLrg : STRING :=
  "Width of Truth/State table is too large. Extra elements are ignored.";
CONSTANT MsgTabResSml : STRING :=
  "Result of Truth/State table has too many elements.";
CONSTANT MsgTabResLrg : STRING :=
  "Result of Truth/State table has too few elements.";

CONSTANT MsgUnknown : STRING :=
  "Unknown error message.";

--------------------------------------------------------------------
-- LOCAL Utilities
--------------------------------------------------------------------
FUNCTION VitalMessage (
        CONSTANT ErrorId : IN VitalErrorType
      ) RETURN STRING IS
BEGIN
    CASE ErrorId IS
        WHEN ErrNegDel    => RETURN MsgNegDel;
        WHEN ErrInpSym    => RETURN MsgInpSym;
        WHEN ErrOutSym    => RETURN MsgOutSym;
        WHEN ErrStaSym    => RETURN MsgStaSym;
        WHEN ErrVctLng    => RETURN MsgVctLng;
        WHEN ErrTabWidSml => RETURN MsgTabWidSml;
        WHEN ErrTabWidLrg => RETURN MsgTabWidLrg;
        WHEN ErrTabResSml => RETURN MsgTabResSml;
        WHEN ErrTabResLrg => RETURN MsgTabResLrg;
        WHEN OTHERS       => RETURN MsgUnknown;
    END CASE;
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId)
      SEVERITY VitalErrorSeverity(ErrorId);
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType;
        CONSTANT Info    : IN STRING
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId) & Info
      SEVERITY VitalErrorSeverity(ErrorId);
END;

PROCEDURE VitalError (
        CONSTANT Routine : IN STRING;
        CONSTANT ErrorId : IN VitalErrorType;
        CONSTANT Info    : IN CHARACTER
```

 Copyright © 2001 IEEE. All rights reserved.

```
) IS
BEGIN
    ASSERT FALSE
      REPORT Routine & ": " & VitalMessage(ErrorId) & Info
      SEVERITY VitalErrorSeverity(ErrorId);
END;

---------------------------------------------------------------------------
PROCEDURE ReportGlitch (
        CONSTANT GlitchRoutine  : IN  STRING;
        CONSTANT OutSignalName  : IN  STRING;
        CONSTANT PreemptedTime  : IN  TIME;
        CONSTANT PreemptedValue : IN  std_ulogic;
        CONSTANT NewTime        : IN  TIME;
        CONSTANT NewValue       : IN  std_ulogic;
        CONSTANT Index          : IN  INTEGER := 0;
        CONSTANT IsArraySignal  : IN  BOOLEAN := FALSE;
        CONSTANT MsgSeverity    : IN  SEVERITY_LEVEL := WARNING
) IS

    VARIABLE StrPtr1, StrPtr2, StrPtr3, StrPtr4, StrPtr5 : LINE;
BEGIN

    Write (StrPtr1, PreemptedTime );
    Write (StrPtr2, NewTime);
    Write (StrPtr3, LogicCvtTable(PreemptedValue));
    Write (StrPtr4, LogicCvtTable(NewValue));
    IF IsArraySignal THEN
        Write (StrPtr5, STRING'( "(" ) );
        Write (StrPtr5, Index);
        Write (StrPtr5, STRING'( ")" ) );
    ELSE
        Write (StrPtr5, STRING'( " " ) );
    END IF;

    -- Issue Report only if Preemted value has not been
    --  removed from event queue
    ASSERT PreemptedTime >  NewTime
      REPORT GlitchRoutine & ": GLITCH Detected on port " &
             OutSignalName & StrPtr5.ALL &
             "; Preempted Future Value := " & StrPtr3.ALL &
             " @ " & StrPtr1.ALL &
             "; Newly Scheduled Value := " & StrPtr4.ALL &
             " @ " & StrPtr2.ALL &
             ";"
      SEVERITY MsgSeverity;

    DEALLOCATE(StrPtr1);
    DEALLOCATE(StrPtr2);
    DEALLOCATE(StrPtr3);
    DEALLOCATE(StrPtr4);
    DEALLOCATE(StrPtr5);
    RETURN;
END ReportGlitch;

---------------------------------------------------------------------------
-- Procedure  : VitalGlitchOnEvent
--            :
-- Parameters : OutSignal ........ signal being driven
--            : OutSignalName..... name of the driven signal
--            : GlitchData........ internal data required by the procedure
--            : NewValue.......... new value being assigned
--            : NewDelay.......... Delay accompanying the assignment
--            :                    (Note: for vectors, this is an array)
--            : GlitchMode........ Glitch generation mode
--            :                     MessagePlusX, MessageOnly,
--            :                     XOnly, NoGlitch )
--            : GlitchDelay....... if <= 0 ns , then there will be no Glitch
--            :                    if >  NewDelay, then there is no Glitch,
--            :                    otherwise, this is the time when a FORCED
--            :                    generation of a glitch will occur.
----------------------------------------------------------------------------
PROCEDURE VitalGlitchOnEvent (
        SIGNAL   OutSignal        : OUT   std_logic;
        CONSTANT OutSignalName    : IN    STRING;
        VARIABLE GlitchData       : INOUT GlitchDataType;
        CONSTANT NewValue         : IN    std_logic;
        CONSTANT NewDelay         : IN    TIME := 0 ns;
        CONSTANT GlitchMode       : IN    VitalGlitchModeType := MessagePlusX;
        CONSTANT GlitchDelay      : IN    TIME := -1 ns;  -- IR#223
        CONSTANT MsgSeverity      : IN    SEVERITY_LEVEL := WARNING
```

Copyright © 2001 IEEE. All rights reserved.

```
) IS
-- --------------------------------------------------------------------------
    VARIABLE NoGlitchDet  : BOOLEAN := FALSE;
    VARIABLE OldGlitch    : BOOLEAN := FALSE;
    VARIABLE Dly          : TIME    := NewDelay;

BEGIN
    -- If nothing to schedule, just return
    IF NewDelay < 0 ns THEN
        IF (NewValue /= GlitchData.SchedValue) THEN
            VitalError ( "VitalGlitchOnEvent", ErrNegDel, OutSignalName );
        END IF;

    ELSE
        -- If nothing currently scheduled
        IF GlitchData.SchedTime <= NOW THEN
            GlitchData.CurrentValue := GlitchData.SchedValue;
            IF (GlitchDelay <= 0 ns) THEN
                IF (NewValue = GlitchData.SchedValue) THEN RETURN; END IF;
                NoGlitchDet := TRUE;
            END IF;

        -- Transaction currently scheduled - if glitch already happened
        ELSIF GlitchData.GlitchTime <= NOW THEN
            GlitchData.CurrentValue := 'X';
            OldGlitch := TRUE;
            IF (GlitchData.SchedValue = NewValue) THEN
                dly := Minimum( GlitchData.SchedTime-NOW, NewDelay );
            END IF;

        -- Transaction currently scheduled (no glitch if same value)
        ELSIF (GlitchData.SchedValue = NewValue) AND
              (GlitchData.SchedTime = GlitchData.GlitchTime) AND
              (GlitchDelay <= 0 ns) THEN
            NoGlitchDet := TRUE;
            Dly := Minimum( GlitchData.SchedTime-NOW, NewDelay );

        END IF;

        GlitchData.SchedTime := NOW+Dly;
        IF OldGlitch THEN
            OutSignal <= NewValue AFTER Dly;

        ELSIF NoGlitchDet THEN
            GlitchData.GlitchTime := NOW+Dly;
            OutSignal <= NewValue AFTER Dly;

        ELSE -- new glitch
            GlitchData.GlitchTime := GlitchMinTime ( GlitchData.GlitchTime,
                                                     NOW+GlitchDelay );

            IF (GlitchMode = MessagePlusX) OR
               (GlitchMode = MessageOnly) THEN
                ReportGlitch ( "VitalGlitchOnEvent", OutSignalName,
                               GlitchData.GlitchTime, GlitchData.SchedValue,
                               (Dly + NOW), NewValue,
                               MsgSeverity=>MsgSeverity );
            END IF;

            IF (GlitchMode = MessagePlusX) OR (GlitchMode = XOnly) THEN
                OutSignal <= 'X' AFTER GlitchData.GlitchTime-NOW;
                OutSignal <=  TRANSPORT NewValue AFTER Dly;
            ELSE
                OutSignal <= NewValue AFTER Dly;
            END IF;
        END IF;

        GlitchData.SchedValue := NewValue;
    END IF;

    RETURN;
END;

----------------------------------------------------------------------------
PROCEDURE VitalGlitchOnEvent (
        SIGNAL   OutSignal        : OUT   std_logic_vector;
        CONSTANT OutSignalName    : IN    STRING;
        VARIABLE GlitchData       : INOUT GlitchDataArrayType;
        CONSTANT NewValue         : IN    std_logic_vector;
        CONSTANT NewDelay         : IN    VitalTimeArray;
        CONSTANT GlitchMode       : IN    VitalGlitchModeType := MessagePlusX;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT GlitchDelay      : IN    VitalTimeArray;
        CONSTANT MsgSeverity      : IN    SEVERITY_LEVEL := WARNING
) IS

    ALIAS GlDataAlias  : GlitchDataArrayType(1 TO GlitchData'LENGTH)
                            IS GlitchData;
    ALIAS NewValAlias  : std_logic_vector(1 TO NewValue'LENGTH) IS NewValue;
    ALIAS GlDelayAlias : VitalTimeArray(1 TO GlitchDelay'LENGTH)
          IS GlitchDelay;
    ALIAS NewDelAlias  : VitalTimeArray(1 TO NewDelay'LENGTH) IS NewDelay;

    VARIABLE Index       : INTEGER := OutSignal'LEFT;
    VARIABLE Direction   : INTEGER;
    VARIABLE NoGlitchDet : BOOLEAN;
    VARIABLE OldGlitch   : BOOLEAN;
    VARIABLE Dly, GlDly  : TIME;

BEGIN
    IF (OutSignal'LEFT > OutSignal'RIGHT) THEN
        Direction := -1;
    ELSE
        Direction := 1;
    END IF;

    IF ( (OutSignal'LENGTH /=  GlitchData'LENGTH) OR
         (OutSignal'LENGTH /=    NewValue'LENGTH) OR
         (OutSignal'LENGTH /=    NewDelay'LENGTH) OR
         (OutSignal'LENGTH /= GlitchDelay'LENGTH) ) THEN
      VitalError ( "VitalGlitchOnEvent", ErrVctLng, OutSignalName );
      RETURN;
    END IF;

    -- a call to the scalar function cannot be made since the actual
    -- name associated with a signal parameter must be locally static
  FOR n IN 1 TO OutSignal'LENGTH LOOP

    NoGlitchDet := FALSE;
    OldGlitch   := FALSE;
    Dly := NewDelAlias(n);

    -- If nothing to schedule, just skip to next loop iteration
    IF NewDelAlias(n) < 0 ns THEN
        IF (NewValAlias(n) /=  GlDataAlias(n).SchedValue) THEN
            VitalError ( "VitalGlitchOnEvent", ErrNegDel, OutSignalName );
        END IF;
    ELSE
        -- If nothing currently scheduled (i.e. last scheduled
        -- transaction already occurred)
        IF GlDataAlias(n).SchedTime <= NOW THEN
            GlDataAlias(n).CurrentValue := GlDataAlias(n).SchedValue;
            IF (GlDelayAlias(n) <= 0 ns) THEN
                -- Next iteration if no change in value
                IF (NewValAlias(n) = GlDataAlias(n).SchedValue) THEN
                    Index := Index + Direction;
                    NEXT;
                END IF;
                -- since last transaction already occurred there is no glitch
                NoGlitchDet := TRUE;
            END IF;

        -- Transaction currently scheduled - if glitch already happened
        ELSIF GlDataAlias(n).GlitchTime <= NOW THEN
            GlDataAlias(n).CurrentValue := 'X';
            OldGlitch := TRUE;
            IF (GlDataAlias(n).SchedValue = NewValAlias(n)) THEN
                dly := Minimum( GlDataAlias(n).SchedTime-NOW,
                                NewDelAlias(n) );
            END IF;

        -- Transaction currently scheduled
        ELSIF (GlDataAlias(n).SchedValue = NewValAlias(n)) AND
              (GlDataAlias(n).SchedTime = GlDataAlias(n).GlitchTime) AND
              (GlDelayAlias(n) <= 0 ns) THEN
              NoGlitchDet := TRUE;
              Dly := Minimum( GlDataAlias(n).SchedTime-NOW,
                              NewDelAlias(n) );
        END IF;

        -- update last scheduled transaction
        GlDataAlias(n).SchedTime := NOW+Dly;
```

Copyright © 2001 IEEE. All rights reserved.

```
        IF OldGlitch THEN
            OutSignal(Index) <= NewValAlias(n) AFTER Dly;
        ELSIF NoGlitchDet THEN
            -- if no glitch then update last glitch time
            -- and OutSignal(actual_index)
            GlDataAlias(n).GlitchTime := NOW+Dly;
            OutSignal(Index) <= NewValAlias(n) AFTER Dly;
        ELSE   -- new glitch
            GlDataAlias(n).GlitchTime := GlitchMinTime (
                                              GlDataAlias(n).GlitchTime,
                                              NOW+GlDelayAlias(n) );

            IF (GlitchMode = MessagePlusX) OR
               (GlitchMode = MessageOnly) THEN
                ReportGlitch ( "VitalGlitchOnEvent", OutSignalName,
                               GlDataAlias(n).GlitchTime,
                               GlDataAlias(n).SchedValue,
                               (Dly + NOW), NewValAlias(n),
                               Index, TRUE, MsgSeverity );
            END IF;

            IF (GlitchMode = MessagePlusX) OR (GlitchMode = XOnly) THEN
                GlDly := GlDataAlias(n).GlitchTime - NOW;
                OutSignal(Index) <= 'X' AFTER GlDly;
                OutSignal(Index) <= TRANSPORT NewValAlias(n) AFTER Dly;
            ELSE
                OutSignal(Index) <= NewValAlias(n) AFTER Dly;
            END IF;

        END IF; -- glitch / no-glitch
        GlDataAlias(n).SchedValue := NewValAlias(n);

    END IF; -- NewDelAlias(n) < 0 ns
    Index := Index + Direction;
  END LOOP;

    RETURN;
END;

---------------------------------------------------------------------------
-- ------------------------------------------------------------------------
--  PROCEDURE NAME  :  TruthOutputX01Z
--
--  PARAMETERS      :  table_out - output of table
--                     X01Zout   - output converted to X01Z
--                     err       - true if illegal character is encountered
--
--
--  DESCRIPTION     :  converts the output of a truth table to a valid
--                     std_ulogic
-- ------------------------------------------------------------------------
PROCEDURE TruthOutputX01Z (
        CONSTANT TableOut : IN VitalTruthSymbolType;
        VARIABLE X01Zout   : OUT std_ulogic;
        VARIABLE Err       : OUT BOOLEAN
) IS
    VARIABLE TempOut : std_ulogic;
BEGIN
    Err := FALSE;
    TempOut := TruthTableOutMap(TableOut);
    IF (TempOut = '-' ) THEN
        Err := TRUE;
        TempOut := 'X';
        VitalError ( "VitalTruthTable", ErrOutSym, To_TruthChar(TableOut));
    END IF;
    X01Zout := TempOut;
END;

-- ------------------------------------------------------------------------
--  PROCEDURE NAME  :  StateOutputX01Z
--
--  PARAMETERS      :  table_out - output of table
--                     prev_out  - previous output value
--                     X01Zout   - output cojnverted to X01Z
--                     err       - true if illegal character is encountered
--
--  DESCRIPTION     :  converts the output of a state table to a
--                     valid std_ulogic
-- ------------------------------------------------------------------------
PROCEDURE StateOutputX01Z (
        CONSTANT TableOut : IN VitalStateSymbolType;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT PrevOut  : IN std_ulogic;
        VARIABLE X01Zout   : OUT std_ulogic;
        VARIABLE Err       : OUT BOOLEAN
) IS
    VARIABLE TempOut : std_ulogic;
BEGIN
    Err := FALSE;
    TempOut := StateTableOutMap(TableOut);
    IF (TempOut = '-' ) THEN
        Err := TRUE;
        TempOut := 'X' ;
        VitalError ( "VitalStateTable", ErrOutSym, To_StateChar(TableOut));
    ELSIF (TempOut = 'W' ) THEN
        TempOut := To_X01Z(PrevOut);
    END IF;
    X01Zout := TempOut;
END;

-- -------------------------------------------------------------------------
-- PROCEDURE NAME:  StateMatch
--
-- PARAMETERS    :  symbol       - symbol from state table
--                  in2          - input from VitalStateTble procedure
--                                 to state table
--                  in2LastValue - previous value of input
--                  state        - false if the symbol is from the input
--                                  portion of the table,
--                                 true if the symbol is from the state
--                                  portion of the table
--                  Err          - true if symbol is not a valid input symbol
--                  ReturnValue  - true if match occurred
--
-- DESCRIPTION   :  This procedure sets ReturnValue to true if in2 matches
--                  symbol (from the state table).  If symbol is an edge
--                  value edge is set to true and in2 and in2LastValue are
--                  checked against symbol.  Err is set to true if symbol
--                  is an invalid value for the input portion of the state
--                  table.
--
-- -------------------------------------------------------------------------
PROCEDURE StateMatch (
        CONSTANT Symbol       : IN VitalStateSymbolType;
        CONSTANT in2          : IN std_ulogic;
        CONSTANT in2LastValue : IN std_ulogic;
        CONSTANT State        : IN BOOLEAN;
        VARIABLE Err          : OUT BOOLEAN;
        VARIABLE ReturnValue  : OUT BOOLEAN
) IS
BEGIN
    IF (State) THEN
        IF (NOT ValidStateTableState(Symbol)) THEN
            VitalError ( "VitalStateTable", ErrStaSym, To_StateChar(Symbol));
            Err := TRUE;
            ReturnValue := FALSE;
        ELSE
            Err := FALSE;
            ReturnValue := StateTableMatch(in2LastValue, in2, Symbol);
        END IF;
    ELSE
        IF (NOT ValidStateTableInput(Symbol) ) THEN
            VitalError ( "VitalStateTable", ErrInpSym, To_StateChar(Symbol));
            Err := TRUE;
            ReturnValue := FALSE;
        ELSE
            ReturnValue := StateTableMatch(in2LastValue, in2, Symbol);
            Err := FALSE;
        END IF;
    END IF;
END;

-- ------------------------------------------------------------------------
-- FUNCTION NAME:  StateTableLookUp
--
-- PARAMETERS   :  StateTable     - state table
--                 PresentDataIn  - current inputs
--                 PreviousDataIn - previous inputs and states
--                 NumStates      - number of state variables
--                 PresentOutputs - current state and current outputs
--
-- DESCRIPTION  :  This function is used to find the output of the
--                 StateTable corresponding to a given set of inputs.
```


Copyright © 2001 IEEE. All rights reserved.

```
--
-- ------------------------------------------------------------------------
FUNCTION StateTableLookUp (
        CONSTANT StateTable     : VitalStateTableType;
        CONSTANT PresentDataIn  : std_logic_vector;
        CONSTANT PreviousDataIn : std_logic_vector;
        CONSTANT NumStates      : NATURAL;
        CONSTANT PresentOutputs : std_logic_vector
      ) RETURN std_logic_vector IS

    CONSTANT InputSize    : INTEGER := PresentDataIn'LENGTH;
    CONSTANT NumInputs    : INTEGER := InputSize + NumStates - 1;
    CONSTANT TableEntries : INTEGER := StateTable'LENGTH(1);
    CONSTANT TableWidth   : INTEGER := StateTable'LENGTH(2);
    CONSTANT OutSize      : INTEGER := TableWidth - InputSize - NumStates;
    VARIABLE Inputs       : std_logic_vector(0 TO NumInputs);
    VARIABLE PrevInputs   : std_logic_vector(0 TO NumInputs)
                            := (OTHERS => 'X');
    VARIABLE ReturnValue  : std_logic_vector(0 TO (OutSize-1))
                            := (OTHERS => 'X');
    VARIABLE Temp   : std_ulogic;
    VARIABLE Match  : BOOLEAN;
    VARIABLE Err    : BOOLEAN := FALSE;

    -- This needs to be done since the TableLookup arrays must be
    -- ascending starting with 0
    VARIABLE TableAlias   : VitalStateTableType(0 TO TableEntries - 1,
                                                0 TO TableWidth - 1)
                          := StateTable;

BEGIN
    Inputs(0 TO InputSize-1) := PresentDataIn;
    Inputs(InputSize TO NumInputs) := PresentOutputs(0 TO NumStates - 1);
    PrevInputs(0 TO InputSize - 1) := PreviousDataIn(0 TO InputSize - 1);

  ColLoop: -- Compare each entry in the table
    FOR i IN TableAlias'RANGE(1) LOOP

    RowLoop: -- Check each element of the entry
      FOR j IN 0 TO InputSize + NumStates  LOOP

        IF (j = InputSize + NumStates) THEN        -- a match occurred
            FOR k IN 0 TO Minimum(OutSize, PresentOutputs'LENGTH)-1  LOOP
                StateOutputX01Z (
                        TableAlias(i, TableWidth - k - 1),
                        PresentOutputs(PresentOutputs'LENGTH - k - 1),
                        Temp, Err);
                ReturnValue(OutSize - k - 1) := Temp;
                IF (Err) THEN
                    ReturnValue := (OTHERS => 'X');
                    RETURN ReturnValue;
                END IF;
            END LOOP;
            RETURN ReturnValue;
        END IF;

        StateMatch ( TableAlias(i,j),
                     Inputs(j), PrevInputs(j),
                     j >= InputSize, Err, Match);
        EXIT RowLoop WHEN NOT(Match);
        EXIT ColLoop WHEN Err;
      END LOOP RowLoop;
    END LOOP ColLoop;

    ReturnValue := (OTHERS => 'X');
    RETURN ReturnValue;
END;

--------------------------------------------------------------------
-- to_ux01z
-------------------------------------------------------------------
FUNCTION To_UX01Z  ( s : std_ulogic
      ) RETURN  UX01Z IS
BEGIN
    RETURN cvt_to_ux01z (s);
END;

---------------------------------------------------------------------------
-- Function  : GetEdge
-- Purpose   : Converts transitions on a given input signal into a
--             enumeration value representing the transition or level
```

 Copyright © 2001 IEEE. All rights reserved.

```
--              of the signal.
--
--    previous "value"   current "value"     :=   "edge"
--   ---------------------------------------------------------
--     '1' | 'H'          '1' | 'H'                  '1'    level, no edge
--     '0' | 'L'          '1' | 'H'                  '/'    rising edge
--      others            '1' | 'H'                  'R'    rising from X
--
--     '1' | 'H'          '0' | 'L'                  '\'    falling egde
--     '0' | 'L'          '0' | 'L'                  '0'    level, no edge
--      others            '0' | 'L'                  'F'    falling from X
--
--     'X' | 'W' | '-'    'X' | 'W' | '-'            'X'    unknown (X) level
--     'Z'                'Z'                        'X'    unknown (X) level
--     'U'                'U'                        'U'    'U' level
--
--     '1' | 'H'           others                    'f'    falling to X
--     '0' | 'L'           others                    'r'    rising to X
--     'X' | 'W' | '-'    'U' | 'Z'                  'x'    unknown (X) edge
--     'Z'                'X' | 'W' | '-' | 'U'      'x'    unknown (X) edge
--     'U'                'X' | 'W' | '-' | 'Z'      'x'    unknown (X) edge
--
---------------------------------------------------------------------------
FUNCTION GetEdge (
        SIGNAL      s : IN    std_logic
      ) RETURN EdgeType IS
BEGIN
    IF (s'EVENT)
        THEN RETURN LogicToEdge  ( s'LAST_VALUE, s );
        ELSE RETURN LogicToLevel ( s );
    END IF;
END;

---------------------------------------------------------------------------
PROCEDURE GetEdge (
        SIGNAL       s : IN    std_logic_vector;
        VARIABLE LastS : INOUT std_logic_vector;
        VARIABLE  Edge :   OUT EdgeArray ) IS

    ALIAS     sAlias : std_logic_vector ( 1 TO     s'LENGTH ) IS s;
    ALIAS LastSAlias : std_logic_vector ( 1 TO LastS'LENGTH ) IS LastS;
    ALIAS  EdgeAlias : EdgeArray ( 1 TO  Edge'LENGTH ) IS Edge;
BEGIN
    IF s'LENGTH /= LastS'LENGTH OR
       s'LENGTH /=  Edge'LENGTH THEN
        VitalError ( "GetEdge", ErrVctLng, "s, LastS, Edge" );
    END IF;

    FOR n IN 1 TO s'LENGTH LOOP
        EdgeAlias(n)  := LogicToEdge( LastSAlias(n), sAlias(n) );
        LastSAlias(n) := sAlias(n);
    END LOOP;
END;

---------------------------------------------------------------------------
FUNCTION  ToEdge     ( Value         : IN std_logic
      ) RETURN EdgeType IS
BEGIN
    RETURN LogicToLevel( Value );
END;

-- Note: This function will likely be replaced by S'DRIVING_VALUE in VHDL'92
----------------------------------------------------------------------------
IMPURE FUNCTION CurValue (
        CONSTANT GlitchData : IN  GlitchDataType
      ) RETURN std_logic IS
BEGIN
    IF NOW >= GlitchData.SchedTime THEN
        RETURN GlitchData.SchedValue;
    ELSIF NOW >= GlitchData.GlitchTime THEN
        RETURN 'X';
    ELSE
        RETURN GlitchData.CurrentValue;
    END IF;
END;
---------------------------------------------------------------------------
IMPURE FUNCTION CurValue (
        CONSTANT GlitchData : IN  GlitchDataArrayType
      ) RETURN std_logic_vector IS
    VARIABLE Result : std_logic_vector(GlitchData'RANGE);
BEGIN
```

```
    FOR n IN GlitchData'RANGE LOOP
        IF NOW >= GlitchData(n).SchedTime THEN
            Result(n) := GlitchData(n).SchedValue;
        ELSIF NOW >= GlitchData(n).GlitchTime THEN
            Result(n) := 'X';
        ELSE
            Result(n) := GlitchData(n).CurrentValue;
        END IF;
    END LOOP;
    RETURN Result;
END;

---------------------------------------------------------------------------
-- function calculation utilities
---------------------------------------------------------------------------

---------------------------------------------------------------------------
-- Function   : VitalSame
-- Returns    : VitalSame compares the state (UX01) of two logic value. A
--              value of 'X' is returned if the values are different.  The
--              common value is returned if the values are equal.
-- Purpose    : When the result of a logic model may be either of two
--              separate input values (eg. when the select on a MUX is 'X'),
--              VitalSame may be used to determine if the result needs to
--              be 'X'.
-- Arguments  : See the declarations below...
---------------------------------------------------------------------------
FUNCTION VitalSame (
        CONSTANT a, b : IN std_ulogic
      ) RETURN std_ulogic IS
BEGIN
    IF To_UX01(a) = To_UX01(b)
        THEN RETURN To_UX01(a);
        ELSE RETURN 'X';
    END IF;
END;

---------------------------------------------------------------------------
-- delay selection utilities
---------------------------------------------------------------------------

---------------------------------------------------------------------------
-- Procedure  : BufPath, InvPath
--
-- Purpose    : BufPath and InvPath compute output change times, based on
--              a change on an input port. The computed output change times
--              returned in the composite parameter 'schd'.
--
--              BufPath and InpPath are used together with the delay path
--              selection functions (GetSchedDelay, VitalAND, VitalOR... )
--              The 'schd' value from each of the input ports of a model are
--              combined by the delay selection functions (VitalAND,
--              VitalOR, ...). The GetSchedDelay procedure converts the
--              combined output changes times to the single delay (delta
--              time) value for scheduling the output change (passed to
--              VitalGlitchOnEvent).
--
--              The values in 'schd' are: (absolute times)
--                inp0  :  time of output change due to input change to 0
--                inp1  :  time of output change due to input change to 1
--                inpX  :  time of output change due to input change to X
--                glch0 :  time of output glitch due to input change to 0
--                glch1 :  time of output glitch due to input change to 1
--
--              The output times are computed from the model INPUT value
--              and not the final value.  For this reason, 'BufPath' should
--              be used to compute the output times for a non-inverting
--              delay paths and 'InvPath' should be used to compute the
--              ouput times for inverting delay paths. Delay paths which
--              include both non-inverting and paths require usage of both
--              'BufPath' and 'InvPath'. (IE this is needed for the
--              select->output path of a MUX -- See the VitalMUX model).
--
--
-- Parameters : schd....... Computed output result times. (INOUT parameter
--                          modified only on input edges)
--              Iedg....... Input port edge/level value.
--               tpd....... Propagation delays from this input
--
---------------------------------------------------------------------------
```

 Copyright © 2001 IEEE. All rights reserved.

```
PROCEDURE BufPath (
        VARIABLE Schd : INOUT SchedType;
        CONSTANT Iedg : IN    EdgeType;
        CONSTANT  tpd : IN    VitalDelayType01
) IS
BEGIN
  CASE Iedg IS
    WHEN '0'|'1'  => NULL;                   -- no edge: no timing update
    WHEN '/'|'R'  => Schd.inp0 := TIME'HIGH;
                     Schd.inp1 := NOW + tpd(tr01);  Schd.Glch1 := Schd.inp1;
                     Schd.InpX := Schd.inp1;
    WHEN '\'|'F'  => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := NOW + tpd(tr10);  Schd.Glch0 := Schd.inp0;
                     Schd.InpX := Schd.inp0;
    WHEN 'r'      => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := TIME'HIGH;
                     Schd.InpX := NOW + tpd(tr01);
    WHEN 'f'      => Schd.inp0 := TIME'HIGH;
                     Schd.inp1 := TIME'HIGH;
                     Schd.InpX := NOW + tpd(tr10);
    WHEN 'x'      => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := TIME'HIGH;
                     -- update for X->X change
                     Schd.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
    WHEN OTHERS   => NULL;                   -- no timing change
  END CASE;
END;

PROCEDURE BufPath (
        VARIABLE Schd : INOUT SchedArray;
        CONSTANT Iedg : IN    EdgeArray;
        CONSTANT  tpd : IN    VitalDelayArrayType01
) IS
BEGIN
  FOR n IN Schd'RANGE LOOP
    CASE Iedg(n) IS
      WHEN '0'|'1'  => NULL;                   -- no edge: no timing update
      WHEN '/'|'R'  => Schd(n).inp0 := TIME'HIGH;
                       Schd(n).inp1 := NOW + tpd(n)(tr01);
                       Schd(n).Glch1 := Schd(n).inp1;
                       Schd(n).InpX := Schd(n).inp1;
      WHEN '\'|'F'  => Schd(n).inp1 := TIME'HIGH;
                       Schd(n).inp0 := NOW + tpd(n)(tr10);
                       Schd(n).Glch0 := Schd(n).inp0;
                       Schd(n).InpX := Schd(n).inp0;
      WHEN 'r'      => Schd(n).inp1 := TIME'HIGH;
                       Schd(n).inp0 := TIME'HIGH;
                       Schd(n).InpX := NOW + tpd(n)(tr01);
      WHEN 'f'      => Schd(n).inp0 := TIME'HIGH;
                       Schd(n).inp1 := TIME'HIGH;
                       Schd(n).InpX := NOW + tpd(n)(tr10);
      WHEN 'x'      => Schd(n).inp1 := TIME'HIGH;
                       Schd(n).inp0 := TIME'HIGH;
                       -- update for X->X change
                       Schd(n).InpX := NOW + Minimum ( tpd(n)(tr10),
                                                       tpd(n)(tr01) );
      WHEN OTHERS   => NULL;                   -- no timing change
    END CASE;
  END LOOP;
END;

PROCEDURE InvPath (
        VARIABLE Schd : INOUT SchedType;
        CONSTANT Iedg : IN    EdgeType;
        CONSTANT  tpd : IN    VitalDelayType01
) IS
BEGIN
  CASE Iedg IS
    WHEN '0'|'1'  => NULL;                   -- no edge: no timing update
    WHEN '/'|'R'  => Schd.inp0 := TIME'HIGH;
                     Schd.inp1 := NOW + tpd(tr10);  Schd.Glch1 := Schd.inp1;
                     Schd.InpX := Schd.inp1;
    WHEN '\'|'F'  => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := NOW + tpd(tr01);  Schd.Glch0 := Schd.inp0;
                     Schd.InpX := Schd.inp0;
    WHEN 'r'      => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := TIME'HIGH;
                     Schd.InpX := NOW + tpd(tr10);
    WHEN 'f'      => Schd.inp0 := TIME'HIGH;
                     Schd.inp1 := TIME'HIGH;
                     Schd.InpX := NOW + tpd(tr01);
```

Copyright © 2001 IEEE. All rights reserved.

```
    WHEN 'x'      => Schd.inp1 := TIME'HIGH;
                     Schd.inp0 := TIME'HIGH;
                     -- update for X->X change
                     Schd.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
    WHEN OTHERS  => NULL;                    -- no timing change
  END CASE;
END;

PROCEDURE InvPath (
        VARIABLE Schd : INOUT SchedArray;
        CONSTANT Iedg : IN    EdgeArray;
        CONSTANT  tpd : IN    VitalDelayArrayType01
) IS
BEGIN
  FOR n IN Schd'RANGE LOOP
    CASE Iedg(n) IS
      WHEN '0'|'1'  => NULL;                   -- no edge: no timing update
      WHEN '/'|'R'  => Schd(n).inp0 := TIME'HIGH;
                       Schd(n).inp1 := NOW + tpd(n)(tr10);
                       Schd(n).Glch1 := Schd(n).inp1;
                       Schd(n).InpX := Schd(n).inp1;
      WHEN '\'|'F'  => Schd(n).inp1 := TIME'HIGH;
                       Schd(n).inp0 := NOW + tpd(n)(tr01);
                       Schd(n).Glch0 := Schd(n).inp0;
                       Schd(n).InpX := Schd(n).inp0;
      WHEN 'r'      => Schd(n).inp1 := TIME'HIGH;
                       Schd(n).inp0 := TIME'HIGH;
                       Schd(n).InpX := NOW + tpd(n)(tr10);
      WHEN 'f'      => Schd(n).inp0 := TIME'HIGH;
                       Schd(n).inp1 := TIME'HIGH;
                       Schd(n).InpX := NOW + tpd(n)(tr01);
      WHEN 'x'      => Schd(n).inp1 := TIME'HIGH;
                       Schd(n).inp0 := TIME'HIGH;
                       -- update for X->X change
                       Schd(n).InpX := NOW + Minimum ( tpd(n)(tr10),
                                                        tpd(n)(tr01) );
      WHEN OTHERS  => NULL;                    -- no timing change
    END CASE;
  END LOOP;
END;

---------------------------------------------------------------------------
-- Procedure  : BufEnab, InvEnab
--
-- Purpose    : BufEnab and InvEnab compute output change times, from a
--              change on an input enable port for a 3-state driver. The
--              computed output change times are returned in the composite
--              parameters 'schd1', 'schd0'.
--
--              BufEnab and InpEnab are used together with the delay path
--              selection functions (GetSchedDelay, VitalAND, VitalOR... )
--              The 'schd' value from each of the non-enable input ports of
--              a model (See BufPath, InvPath) are combined using the delay
--              selection functions (VitalAND,  VitalOR, ...). The
--              GetSchedDelay procedure combines the output times on the
--              enable path with the output times from the data path(s) and
--              computes the single delay (delta time) value for scheduling
--              the output change (passed to VitalGlitchOnEvent)
--
--              The values in 'schd*' are: (absolute times)
--                inp0  :  time of output change due to input change to 0
--                inp1  :  time of output change due to input change to 1
--                inpX  :  time of output change due to input change to X
--                glch0 :  time of output glitch due to input change to 0
--                glch1 :  time of output glitch due to input change to 1
--
--              'schd1'  contains output times for 1->Z, Z->1 transitions.
--              'schd0'  contains output times for 0->Z, Z->0 transitions.
--
--              'BufEnab' is used for computing the output times for an
--              high asserted enable (output 'Z' for enable='0').
--              'InvEnab' is used for computing the output times for an
--              low asserted enable (output 'Z' for enable='1').
--
--              Note: separate 'schd1', 'schd0' parameters are generated
--                    so that the combination of the delay paths from
--                    multiple enable signals may be combined using the
--                    same functions/operators used in combining separate
--                    data paths. (See exampe 2 below)
--
--
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- Parameters : schd1...... Computed output result times for 1->Z, Z->1
--                          transitions. This parameter is modified only on
--                          input edge values (events).
--              schd0...... Computed output result times for 0->Z, 0->1
--                          transitions. This parameter is modified only on
--                          input edge values (events).
--              Iedg....... Input port edge/level value.
--               tpd....... Propagation delays for the enable -> output path.
--
---------------------------------------------------------------------------
PROCEDURE BufEnab (
        VARIABLE Schd1 : INOUT SchedType;
        VARIABLE Schd0 : INOUT SchedType;
        CONSTANT  Iedg : IN    EdgeType;
        CONSTANT   tpd : IN    VitalDelayType01Z
) IS
BEGIN
  CASE Iedg IS
    WHEN ‘0’|’1’  => NULL;                   -- no edge: no timing update
    WHEN ‘/’|’R’  => Schd1.inp0 := TIME’HIGH;
                     Schd1.inp1 := NOW + tpd(trz1);
                     Schd1.Glch1 := Schd1.inp1;
                     Schd1.InpX := Schd1.inp1;
                     Schd0.inp0 := TIME’HIGH;
                     Schd0.inp1 := NOW + tpd(trz0);
                     Schd0.Glch1 := Schd0.inp1;
                     Schd0.InpX := Schd0.inp1;
    WHEN ‘\’|’F’  => Schd1.inp1 := TIME’HIGH;
                     Schd1.inp0 := NOW + tpd(tr1z);
                     Schd1.Glch0 := Schd1.inp0;
                     Schd1.InpX := Schd1.inp0;
                     Schd0.inp1 := TIME’HIGH;
                     Schd0.inp0 := NOW + tpd(tr0z);
                     Schd0.Glch0 := Schd0.inp0;
                     Schd0.InpX := Schd0.inp0;
    WHEN ‘r’      => Schd1.inp1 := TIME’HIGH;
                     Schd1.inp0 := TIME’HIGH;
                     Schd1.InpX := NOW + tpd(trz1);
                     Schd0.inp1 := TIME’HIGH;
                     Schd0.inp0 := TIME’HIGH;
                     Schd0.InpX := NOW + tpd(trz0);
    WHEN ‘f’      => Schd1.inp0 := TIME’HIGH;
                     Schd1.inp1 := TIME’HIGH;
                     Schd1.InpX := NOW + tpd(tr1z);
                     Schd0.inp0 := TIME’HIGH;
                     Schd0.inp1 := TIME’HIGH;
                     Schd0.InpX := NOW + tpd(tr0z);
    WHEN ‘x’      => Schd1.inp0 := TIME’HIGH;
                     Schd1.inp1 := TIME’HIGH;
                     Schd1.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
                     Schd0.inp0 := TIME’HIGH;
                     Schd0.inp1 := TIME’HIGH;
                     Schd0.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
    WHEN OTHERS   => NULL;                   -- no timing change
  END CASE;
END;

PROCEDURE InvEnab (
        VARIABLE Schd1 : INOUT SchedType;
        VARIABLE Schd0 : INOUT SchedType;
        CONSTANT  Iedg : IN    EdgeType;
        CONSTANT   tpd : IN    VitalDelayType01Z
) IS
BEGIN
  CASE Iedg IS
    WHEN ‘0’|’1’  => NULL;                   -- no edge: no timing update
    WHEN ‘/’|’R’  => Schd1.inp0 := TIME’HIGH;
                     Schd1.inp1 := NOW + tpd(tr1z);
                     Schd1.Glch1 := Schd1.inp1;
                     Schd1.InpX := Schd1.inp1;
                     Schd0.inp0 := TIME’HIGH;
                     Schd0.inp1 := NOW + tpd(tr0z);
                     Schd0.Glch1 := Schd0.inp1;
                     Schd0.InpX := Schd0.inp1;
    WHEN ‘\’|’F’  => Schd1.inp1 := TIME’HIGH;
                     Schd1.inp0 := NOW + tpd(trz1);
                     Schd1.Glch0 := Schd1.inp0;
                     Schd1.InpX := Schd1.inp0;
                     Schd0.inp1 := TIME’HIGH;
                     Schd0.inp0 := NOW + tpd(trz0);
                     Schd0.Glch0 := Schd0.inp0;
```

Copyright © 2001 IEEE. All rights reserved.

```
                    Schd0.InpX := Schd0.inp0;
    WHEN 'r'      => Schd1.inp1 := TIME'HIGH;
                    Schd1.inp0 := TIME'HIGH;
                    Schd1.InpX := NOW + tpd(tr1z);
                    Schd0.inp1 := TIME'HIGH;
                    Schd0.inp0 := TIME'HIGH;
                    Schd0.InpX := NOW + tpd(tr0z);
    WHEN 'f'      => Schd1.inp0 := TIME'HIGH;
                    Schd1.inp1 := TIME'HIGH;
                    Schd1.InpX := NOW + tpd(trz1);
                    Schd0.inp0 := TIME'HIGH;
                    Schd0.inp1 := TIME'HIGH;
                    Schd0.InpX := NOW + tpd(trz0);
    WHEN 'x'      => Schd1.inp0 := TIME'HIGH;
                    Schd1.inp1 := TIME'HIGH;
                    Schd1.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
                    Schd0.inp0 := TIME'HIGH;
                    Schd0.inp1 := TIME'HIGH;
                    Schd0.InpX := NOW + Minimum(tpd(tr10),tpd(tr01));
    WHEN OTHERS  => NULL;                    -- no timing change
  END CASE;
END;

---------------------------------------------------------------------------
-- Procedure  : GetSchedDelay
--
-- Purpose    : GetSchedDelay computes the final delay (incremental) for
--              for scheduling an output signal.  The delay is computed
--              from the absolute output times in the 'NewSched' parameter.
--              (See BufPath, InvPath).
--
--              Computation of the output delay for non-3_state outputs
--              consists of selection the appropriate output time based
--              on the new output value 'NewValue' and subtracting 'NOW'
--              to convert to an incremental delay value.
--
--              The Computation of the output delay for 3_state output
--              also includes combination of the enable path delay with
--              the date path delay.
--
-- Parameters : NewDelay... Returned output delay value.
--              GlchDelay.. Returned output delay for the start of a glitch.
--              NewValue... New output value.
--              CurValue... Current value of the output.
--              NewSched... Composite containing the combined absolute
--                          output times from the data inputs.
--              EnSched1... Composite containing the combined absolute
--                          output times from the enable input(s).
--                          (for a 3_state output transitions 1->Z, Z->1)
--              EnSched0... Composite containing the combined absolute
--                          output times from the enable input(s).
--                          (for a 3_state output transitions 0->Z, Z->0)
--
---------------------------------------------------------------------------
PROCEDURE GetSchedDelay (
        VARIABLE   NewDelay : OUT TIME;
        VARIABLE  GlchDelay : OUT TIME;
        CONSTANT   NewValue : IN  std_ulogic;
        CONSTANT   CurValue : IN  std_ulogic;
        CONSTANT   NewSched : IN  SchedType
) IS
    VARIABLE Tim, Glch : TIME;
BEGIN

    CASE To_UX01(NewValue) IS
      WHEN '0'     => Tim  := NewSched.inp0;
                     Glch := NewSched.Glch1;
      WHEN '1'     => Tim  := NewSched.inp1;
                     Glch := NewSched.Glch0;
      WHEN OTHERS => Tim  := NewSched.InpX;
                     Glch := -1 ns;
    END CASE;
    IF (CurValue /= NewValue)
      THEN Glch := -1 ns;
    END IF;

    NewDelay  := Tim  - NOW;
    IF Glch < 0 ns
        THEN GlchDelay := Glch;
        ELSE GlchDelay := Glch - NOW;
    END IF; -- glch < 0 ns
```

 Copyright © 2001 IEEE. All rights reserved.

```
END;

PROCEDURE GetSchedDelay (
        VARIABLE   NewDelay : OUT VitalTimeArray;
        VARIABLE  GlchDelay : OUT VitalTimeArray;
        CONSTANT   NewValue : IN  std_logic_vector;
        CONSTANT   CurValue : IN  std_logic_vector;
        CONSTANT   NewSched : IN  SchedArray
) IS
    VARIABLE Tim, Glch : TIME;
    ALIAS  NewDelayAlias : VitalTimeArray( NewDelay'LENGTH DOWNTO 1)
           IS NewDelay;
    ALIAS GlchDelayAlias : VitalTimeArray(GlchDelay'LENGTH DOWNTO 1)
           IS GlchDelay;
    ALIAS  NewSchedAlias : SchedArray( NewSched'LENGTH DOWNTO 1)
           IS NewSched;
    ALIAS  NewValueAlias : std_logic_vector (  NewValue'LENGTH DOWNTO 1 )
                            IS  NewValue;
    ALIAS  CurValueAlias : std_logic_vector (  CurValue'LENGTH DOWNTO 1 )
                            IS  CurValue;
BEGIN
  FOR n IN NewDelay'LENGTH DOWNTO 1 LOOP
    CASE To_UX01(NewValueAlias(n)) IS
      WHEN '0'     => Tim  := NewSchedAlias(n).inp0;
                      Glch := NewSchedAlias(n).Glch1;
      WHEN '1'     => Tim  := NewSchedAlias(n).inp1;
                      Glch := NewSchedAlias(n).Glch0;
      WHEN OTHERS => Tim  := NewSchedAlias(n).InpX;
                      Glch := -1 ns;
    END CASE;
    IF (CurValueAlias(n) /= NewValueAlias(n))
      THEN Glch := -1 ns;
    END IF;

    NewDelayAlias(n) := Tim  - NOW;
    IF Glch < 0 ns
        THEN GlchDelayAlias(n) := Glch;
        ELSE GlchDelayAlias(n) := Glch - NOW;
    END IF; -- glch < 0 ns
  END LOOP;
  RETURN;
END;

PROCEDURE GetSchedDelay (
        VARIABLE   NewDelay : OUT TIME;
        VARIABLE  GlchDelay : OUT TIME;
        CONSTANT   NewValue : IN  std_ulogic;
        CONSTANT   CurValue : IN  std_ulogic;
        CONSTANT   NewSched : IN  SchedType;
        CONSTANT   EnSched1 : IN  SchedType;
        CONSTANT   EnSched0 : IN  SchedType
) IS
    SUBTYPE v2 IS std_logic_vector(0 TO 1);
    VARIABLE Tim, Glch : TIME;
BEGIN

    CASE v2'(To_X01Z(CurValue) & To_X01Z(NewValue)) IS
      WHEN "00"     => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                       Glch := GlitchMinTime(NewSched.Glch1,EnSched0.Glch0);
      WHEN "01"     => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                       Glch := EnSched1.Glch0;
      WHEN "0Z"     => Tim  := EnSched0.inp0;
                       Glch := NewSched.Glch1;
      WHEN "0X"     => Tim  := Maximum (NewSched.InpX, EnSched1.InpX);
                       Glch := 0 ns;
      WHEN "10"     => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                       Glch := EnSched0.Glch0;
      WHEN "11"     => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                       Glch := GlitchMinTime(NewSched.Glch0,EnSched1.Glch0);
      WHEN "1Z"     => Tim  := EnSched1.inp0;
                       Glch := NewSched.Glch0;
      WHEN "1X"     => Tim  := Maximum (NewSched.InpX, EnSched0.InpX);
                       Glch := 0 ns;
      WHEN "Z0"     => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                       IF NewSched.Glch0 > NOW
                         THEN Glch := Maximum(NewSched.Glch1,EnSched1.inp1);
                         ELSE Glch := 0 ns;
                       END IF;
      WHEN "Z1"     => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                       IF NewSched.Glch1 > NOW
                         THEN Glch := Maximum(NewSched.Glch0,EnSched0.inp1);
```

Copyright © 2001 IEEE. All rights reserved.

```
                        ELSE Glch := 0 ns;
                    END IF;
       WHEN "ZX"     => Tim  := Maximum (NewSched.InpX, EnSched1.InpX);
                        Glch := 0 ns;
       WHEN "ZZ"     => Tim  := Maximum (EnSched1.InpX, EnSched0.InpX);
                        Glch := 0 ns;
       WHEN "X0"     => Tim  := Maximum (NewSched.inp0, EnSched0.inp1);
                        Glch := 0 ns;
       WHEN "X1"     => Tim  := Maximum (NewSched.inp1, EnSched1.inp1);
                        Glch := 0 ns;
       WHEN "XZ"     => Tim  := Maximum (EnSched1.InpX, EnSched0.InpX);
                        Glch := 0 ns;
       WHEN OTHERS   => Tim  := Maximum (NewSched.InpX, EnSched1.InpX);
                        Glch := 0 ns;

    END CASE;
    NewDelay  := Tim  - NOW;
    IF Glch < 0 ns
        THEN GlchDelay := Glch;
        ELSE GlchDelay := Glch - NOW;
    END IF; -- glch < 0 ns
END;

---------------------------------------------------------------------------
-- Operators and Functions for combination (selection) of path delays
-- > These functions support selection of the "appripriate" path delay
--   dependent on the logic function.
-- > These functions only "select" from the possable output times. No
--   calculation (addition) of delays is performed.
-- > See description of 'BufPath', 'InvPath' and 'GetSchedDelay'
-- > See primitive PROCEDURE models for examples.
---------------------------------------------------------------------------

FUNCTION "not"  (
        CONSTANT a : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    z.inp1  := a.inp0 ;
    z.inp0  := a.inp1 ;
    z.InpX  := a.InpX ;
    z.Glch1 := a.Glch0;
    z.Glch0 := a.Glch1;
    RETURN (z);
END;

IMPURE FUNCTION "and"  (
        CONSTANT a, b : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    z.inp1  := Maximum   ( a.inp1 , b.inp1  );
    z.inp0  := Minimum   ( a.inp0 , b.inp0  );
    z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
    z.Glch1 := Maximum   ( a.Glch1, b.Glch1 );
    z.Glch0 := GlitchMinTime ( a.Glch0, b.Glch0 );
    RETURN (z);
END;

IMPURE FUNCTION "or"   (
        CONSTANT a, b : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    z.inp0  := Maximum   ( a.inp0 , b.inp0  );
    z.inp1  := Minimum   ( a.inp1 , b.inp1  );
    z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
    z.Glch0 := Maximum   ( a.Glch0, b.Glch0 );
    z.Glch1 := GlitchMinTime ( a.Glch1, b.Glch1 );
    RETURN (z);
END;

IMPURE FUNCTION "nand" (
        CONSTANT a, b : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    z.inp0  := Maximum   ( a.inp1 , b.inp1  );
    z.inp1  := Minimum   ( a.inp0 , b.inp0  );
    z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
    z.Glch0 := Maximum   ( a.Glch1, b.Glch1 );
```

 Copyright © 2001 IEEE. All rights reserved.

```
    z.Glch1 := GlitchMinTime ( a.Glch0, b.Glch0 );
    RETURN (z);
END;

IMPURE FUNCTION "nor"  (
        CONSTANT a, b : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    z.inp1  := Maximum   ( a.inp0 , b.inp0  );
    z.inp0  := Minimum   ( a.inp1 , b.inp1  );
    z.InpX  := GlitchMinTime ( a.InpX , b.InpX  );
    z.Glch1 := Maximum   ( a.Glch0, b.Glch0 );
    z.Glch0 := GlitchMinTime ( a.Glch1, b.Glch1 );
    RETURN (z);
END;

-- ------------------------------------------------------------------------
-- Delay Calculation for 2-bit Logical gates.
-- ------------------------------------------------------------------------
IMPURE FUNCTION VitalXOR2   (
        CONSTANT ab,ai, bb,bi : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    -- z = (a AND b) NOR (a NOR b)
    z.inp1  :=   Maximum (  Minimum (ai.inp0 , bi.inp0 ),
                            Minimum (ab.inp1 , bb.inp1 ) );
    z.inp0  :=   Minimum (  Maximum (ai.inp1 , bi.inp1 ),
                            Maximum (ab.inp0 , bb.inp0 ) );
    z.InpX  :=   Maximum (  Maximum (ai.InpX , bi.InpX ),
                            Maximum (ab.InpX , bb.InpX ) );
    z.Glch1 :=   Maximum (GlitchMinTime (ai.Glch0, bi.Glch0),
                          GlitchMinTime (ab.Glch1, bb.Glch1) );
    z.Glch0 := GlitchMinTime (  Maximum (ai.Glch1, bi.Glch1),
                            Maximum (ab.Glch0, bb.Glch0) );
    RETURN (z);
END;

IMPURE FUNCTION VitalXNOR2  (
        CONSTANT ab,ai, bb,bi : IN SchedType
      ) RETURN SchedType IS
    VARIABLE z : SchedType;
BEGIN
    -- z = (a AND b) OR (a NOR b)
    z.inp0  :=   Maximum (  Minimum (ab.inp0 , bb.inp0 ),
                            Minimum (ai.inp1 , bi.inp1 ) );
    z.inp1  :=   Minimum (  Maximum (ab.inp1 , bb.inp1 ),
                            Maximum (ai.inp0 , bi.inp0 ) );
    z.InpX  :=   Maximum (  Maximum (ab.InpX , bb.InpX ),
                            Maximum (ai.InpX , bi.InpX ) );
    z.Glch0 :=   Maximum (GlitchMinTime (ab.Glch0, bb.Glch0),
                          GlitchMinTime (ai.Glch1, bi.Glch1) );
    z.Glch1 := GlitchMinTime (  Maximum (ab.Glch1, bb.Glch1),
                            Maximum (ai.Glch0, bi.Glch0) );
    RETURN (z);
END;

-- ------------------------------------------------------------------------
-- Delay Calculation for 3-bit Logical gates.
-- ------------------------------------------------------------------------
IMPURE FUNCTION VitalXOR3   (
        CONSTANT ab,ai, bb,bi, cb,ci : IN SchedType )
  RETURN SchedType IS
BEGIN
    RETURN VitalXOR2 ( VitalXOR2 (ab,ai, bb,bi),
                       VitalXOR2 (ai,ab, bi,bb),
                       cb, ci );
END;

IMPURE FUNCTION VitalXNOR3  (
        CONSTANT ab,ai, bb,bi, cb,ci : IN SchedType )
  RETURN SchedType IS
BEGIN
    RETURN VitalXNOR2 ( VitalXOR2 ( ab,ai, bb,bi ),
                        VitalXOR2 ( ai,ab, bi,bb ),
                        cb, ci );
END;

-- ------------------------------------------------------------------------
-- Delay Calculation for 4-bit Logical gates.
```

Copyright © 2001 IEEE. All rights reserved.

```
    -- ---------------------------------------------------------------------------
    IMPURE FUNCTION VitalXOR4   (
            CONSTANT ab,ai, bb,bi, cb,ci, db,di : IN SchedType )
      RETURN SchedType IS
    BEGIN
        RETURN VitalXOR2 ( VitalXOR2 ( ab,ai, bb,bi ),
                           VitalXOR2 ( ai,ab, bi,bb ),
                           VitalXOR2 ( cb,ci, db,di ),
                           VitalXOR2 ( ci,cb, di,db ) );
    END;

    IMPURE FUNCTION VitalXNOR4  (
            CONSTANT ab,ai, bb,bi, cb,ci, db,di : IN SchedType )
      RETURN SchedType IS
    BEGIN
        RETURN VitalXNOR2 ( VitalXOR2 ( ab,ai, bb,bi ),
                            VitalXOR2 ( ai,ab, bi,bb ),
                            VitalXOR2 ( cb,ci, db,di ),
                            VitalXOR2 ( ci,cb, di,db ) );
    END;

    -- ---------------------------------------------------------------------------
    -- Delay Calculation for N-bit Logical gates.
    -- ---------------------------------------------------------------------------
    -- Note: index range on datab,datai assumed to be 1 TO length.
    --       This is enforced by internal only usage of this Function
    IMPURE FUNCTION VitalXOR   (
            CONSTANT DataB, DataI : IN SchedArray
          ) RETURN SchedType IS
            CONSTANT Leng : INTEGER := DataB'LENGTH;
    BEGIN
        IF Leng = 2 THEN
            RETURN VitalXOR2 ( DataB(1),DataI(1), DataB(2),DataI(2) );
        ELSE
            RETURN VitalXOR2 ( VitalXOR ( DataB(1 TO Leng-1),
                                          DataI(1 TO Leng-1) ),
                               VitalXOR ( DataI(1 TO Leng-1),
                                          DataB(1 TO Leng-1) ),
                               DataB(Leng),DataI(Leng) );
        END IF;
    END;

    -- Note: index range on datab,datai assumed to be 1 TO length.
    --       This is enforced by internal only usage of this Function
    IMPURE FUNCTION VitalXNOR  (
            CONSTANT DataB, DataI : IN SchedArray
          ) RETURN SchedType IS
            CONSTANT Leng : INTEGER := DataB'LENGTH;
    BEGIN
        IF Leng = 2 THEN
            RETURN VitalXNOR2 ( DataB(1),DataI(1), DataB(2),DataI(2) );
        ELSE
            RETURN VitalXNOR2 ( VitalXOR ( DataB(1 TO Leng-1),
                                           DataI(1 TO Leng-1) ),
                                VitalXOR ( DataI(1 TO Leng-1),
                                           DataB(1 TO Leng-1) ),
                                DataB(Leng),DataI(Leng) );
        END IF;
    END;

    -- ---------------------------------------------------------------------------
    -- Multiplexor
    --   MUX   .......... result := data(dselect)
    --   MUX2  .......... 2-input mux; result := data0 when (dselect = '0'),
    --                                           data1 when (dselect = '1'),
    --                        'X' when (dselect = 'X') and (data0 /= data1)
    --   MUX4  .......... 4-input mux; result := data(dselect)
    --   MUX8  .......... 8-input mux; result := data(dselect)
    -- ---------------------------------------------------------------------------
    FUNCTION VitalMUX2  (
            CONSTANT d1, d0 : IN SchedType;
            CONSTANT sb, SI : IN SchedType
          ) RETURN SchedType IS
    BEGIN
        RETURN (d1 AND sb) OR (d0 AND (NOT SI) );
    END;
--
    FUNCTION VitalMUX4  (
            CONSTANT Data : IN SchedArray4;
            CONSTANT sb   : IN SchedArray2;
            CONSTANT SI   : IN SchedArray2
```

 Copyright © 2001 IEEE. All rights reserved.

```
        ) RETURN SchedType IS
    BEGIN
        RETURN    (      sb(1)  AND VitalMUX2(Data(3),Data(2), sb(0), SI(0)) )
               OR ( (NOT SI(1)) AND VitalMUX2(Data(1),Data(0), sb(0), SI(0)) );
    END;

    FUNCTION VitalMUX8  (
            CONSTANT Data : IN SchedArray8;
            CONSTANT sb   : IN SchedArray3;
            CONSTANT SI   : IN SchedArray3
          ) RETURN SchedType IS
    BEGIN
        RETURN    ( (    sb(2)) AND VitalMUX4 (Data(7 DOWNTO 4),
                                   sb(1 DOWNTO 0), SI(1 DOWNTO 0) ) )
               OR ( (NOT SI(2)) AND VitalMUX4 (Data(3 DOWNTO 0),
                                   sb(1 DOWNTO 0), SI(1 DOWNTO 0) ) );
    END;
--
    FUNCTION VInterMux   (
            CONSTANT Data : IN SchedArray;
            CONSTANT sb   : IN SchedArray;
            CONSTANT SI   : IN SchedArray
          ) RETURN SchedType IS
        CONSTANT sMsb : INTEGER := sb'LENGTH;
        CONSTANT dMsbHigh : INTEGER := Data'LENGTH;
        CONSTANT dMsbLow  : INTEGER := Data'LENGTH/2;
    BEGIN
        IF sb'LENGTH = 1 THEN
          RETURN VitalMUX2( Data(2), Data(1), sb(1), SI(1) );
        ELSIF sb'LENGTH = 2 THEN
          RETURN VitalMUX4( Data, sb, SI );
        ELSIF sb'LENGTH = 3 THEN
          RETURN VitalMUX8( Data, sb, SI );
        ELSIF sb'LENGTH > 3 THEN
          RETURN ((    sb(sMsb)) AND VInterMux( Data(dMsbLow  DOWNTO  1),
                                                sb(sMsb-1 DOWNTO 1),
                                                SI(sMsb-1 DOWNTO 1) ))
              OR ((NOT SI(sMsb)) AND VInterMux( Data(dMsbHigh DOWNTO dMsbLow+1),
                                                sb(sMsb-1 DOWNTO 1),
                                                SI(sMsb-1 DOWNTO 1) ));
        ELSE
          RETURN (0 ns, 0 ns, 0 ns, 0 ns, 0 ns); -- dselect'LENGTH < 1
        END IF;
    END;
--
    FUNCTION VitalMUX   (
            CONSTANT Data : IN SchedArray;
            CONSTANT sb   : IN SchedArray;
            CONSTANT SI   : IN SchedArray
          ) RETURN SchedType IS
        CONSTANT msb : INTEGER := 2**sb'LENGTH;
        VARIABLE    lDat : SchedArray(msb DOWNTO 1);
        ALIAS DataAlias : SchedArray ( Data'LENGTH DOWNTO 1 ) IS Data;
        ALIAS   sbAlias : SchedArray (   sb'LENGTH DOWNTO 1 ) IS sb;
        ALIAS   siAlias : SchedArray (   SI'LENGTH DOWNTO 1 ) IS SI;
    BEGIN
        IF Data'LENGTH <= msb THEN
            FOR i IN Data'LENGTH DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
            FOR i IN msb DOWNTO Data'LENGTH+1 LOOP
                lDat(i) := DefSchedAnd;
            END LOOP;
        ELSE
            FOR i IN msb DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
        END IF;
        RETURN VInterMux( lDat, sbAlias, siAlias );
    END;

    -- ------------------------------------------------------------------------
    -- Decoder
    --          General Algorithm :
    --              (a) Result(...) := '0' when (enable = '0')
    --              (b) Result(data) := '1'; all other subelements = '0'
    --              ... Result array is decending (n-1 downto 0)
    --
    --          DECODERn  .......... n:2**n decoder
    -- ------------------------------------------------------------------------
    FUNCTION VitalDECODER2  (
```

```
            CONSTANT DataB  : IN SchedType;
            CONSTANT DataI  : IN SchedType;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        VARIABLE Result : SchedArray2;
    BEGIN
        Result(1) := Enable AND (    DataB);
        Result(0) := Enable AND (NOT DataI);
        RETURN Result;
    END;

    FUNCTION VitalDECODER4  (
            CONSTANT DataB  : IN SchedArray2;
            CONSTANT DataI  : IN SchedArray2;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        VARIABLE Result : SchedArray4;
    BEGIN
        Result(3) := Enable AND (    DataB(1)) AND (    DataB(0));
        Result(2) := Enable AND (    DataB(1)) AND (NOT DataI(0));
        Result(1) := Enable AND (NOT DataI(1)) AND (    DataB(0));
        Result(0) := Enable AND (NOT DataI(1)) AND (NOT DataI(0));
        RETURN Result;
    END;

    FUNCTION VitalDECODER8  (
            CONSTANT DataB  : IN SchedArray3;
            CONSTANT DataI  : IN SchedArray3;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        VARIABLE Result : SchedArray8;
    BEGIN
        Result(7):= Enable AND (    DataB(2))AND(    DataB(1))AND(    DataB(0));
        Result(6):= Enable AND (    DataB(2))AND(    DataB(1))AND(NOT DataI(0));
        Result(5):= Enable AND (    DataB(2))AND(NOT DataI(1))AND(    DataB(0));
        Result(4):= Enable AND (    DataB(2))AND(NOT DataI(1))AND(NOT DataI(0));
        Result(3):= Enable AND (NOT DataI(2))AND(    DataB(1))AND(    DataB(0));
        Result(2):= Enable AND (NOT DataI(2))AND(    DataB(1))AND(NOT DataI(0));
        Result(1):= Enable AND (NOT DataI(2))AND(NOT DataI(1))AND(    DataB(0));
        Result(0):= Enable AND (NOT DataI(2))AND(NOT DataI(1))AND(NOT DataI(0));
        RETURN Result;
    END;


    FUNCTION VitalDECODER   (
            CONSTANT DataB  : IN SchedArray;
            CONSTANT DataI  : IN SchedArray;
            CONSTANT Enable : IN SchedType
          ) RETURN SchedArray IS
        CONSTANT DMsb : INTEGER := DataB'LENGTH - 1;
        ALIAS DataBAlias : SchedArray ( DMsb DOWNTO 0 ) IS DataB;
        ALIAS DataIAlias : SchedArray ( DMsb DOWNTO 0 ) IS DataI;
    BEGIN
        IF DataB'LENGTH = 1 THEN
            RETURN  VitalDECODER2 ( DataBAlias(    0     ),
                                    DataIAlias(    0     ), Enable );
        ELSIF DataB'LENGTH = 2 THEN
            RETURN  VitalDECODER4 ( DataBAlias(1 DOWNTO 0),
                                    DataIAlias(1 DOWNTO 0), Enable );
        ELSIF DataB'LENGTH = 3 THEN
            RETURN  VitalDECODER8 ( DataBAlias(2 DOWNTO 0),
                                    DataIAlias(2 DOWNTO 0), Enable );
        ELSIF DataB'LENGTH > 3 THEN
            RETURN  VitalDECODER  ( DataBAlias(DMsb-1 DOWNTO 0),
                                    DataIAlias(DMsb-1 DOWNTO 0),
                                    Enable AND (    DataBAlias(DMsb)) )
                  & VitalDECODER  ( DataBAlias(DMsb-1 DOWNTO 0),
                                    DataIAlias(DMsb-1 DOWNTO 0),
                                    Enable AND (NOT DataIAlias(DMsb)) );
        ELSE
            RETURN DefSchedArray2;
        END IF;
    END;


-------------------------------------------------------------------------------
-- PRIMITIVES
-------------------------------------------------------------------------------
    -- ------------------------------------------------------------------------
    -- N-bit wide Logical gates.
    -- ------------------------------------------------------------------------
```

 Copyright © 2001 IEEE. All rights reserved.

```
    FUNCTION VitalAND    (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '1';
        FOR i IN Data'RANGE LOOP
            Result := Result AND Data(i);
        END LOOP;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalOR     (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result OR Data(i);
        END LOOP;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalXOR    (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result XOR Data(i);
        END LOOP;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalNAND   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '1';
        FOR i IN Data'RANGE LOOP
            Result := Result AND Data(i);
        END LOOP;
        RETURN ResultMap(NOT Result);
    END;
--
    FUNCTION VitalNOR    (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result OR Data(i);
        END LOOP;
        RETURN ResultMap(NOT Result);
    END;
--
    FUNCTION VitalXNOR   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        Result := '0';
        FOR i IN Data'RANGE LOOP
            Result := Result XOR Data(i);
        END LOOP;
```

Copyright © 2001 IEEE. All rights reserved.

```
        RETURN ResultMap(NOT Result);
    END;

    -- ----------------------------------------------------------------------------
    -- Commonly used 2-bit Logical gates.
    -- ----------------------------------------------------------------------------
    FUNCTION VitalAND2   (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a AND b);
    END;
--
    FUNCTION VitalOR2    (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a OR b);
    END;
--
    FUNCTION VitalXOR2   (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a XOR b);
    END;
--
    FUNCTION VitalNAND2  (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a NAND b);
    END;
--
    FUNCTION VitalNOR2   (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a NOR b);
    END;
--
    FUNCTION VitalXNOR2  (
            CONSTANT       a, b :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a XOR b));
    END;
--
    -- ----------------------------------------------------------------------------
    -- Commonly used 3-bit Logical gates.
    -- ----------------------------------------------------------------------------
    FUNCTION VitalAND3   (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a AND b AND c);
    END;
--
    FUNCTION VitalOR3    (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a OR b OR c);
    END;
--
```

 Copyright © 2001 IEEE. All rights reserved.

```
    FUNCTION VitalXOR3   (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a XOR b XOR c);
    END;
--
    FUNCTION VitalNAND3  (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a AND b AND c));
    END;
--
    FUNCTION VitalNOR3   (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a OR b OR c));
    END;
--
    FUNCTION VitalXNOR3  (
            CONSTANT    a, b, c :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a XOR b XOR c));
    END;

    -- ---------------------------------------------------------------------------
    -- Commonly used 4-bit Logical gates.
    -- ---------------------------------------------------------------------------
    FUNCTION VitalAND4   (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a AND b AND c AND d);
    END;
--
    FUNCTION VitalOR4    (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a OR b OR c OR d);
    END;
--
    FUNCTION VitalXOR4   (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(a XOR b XOR c XOR d);
    END;
--
    FUNCTION VitalNAND4  (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a AND b AND c AND d));
    END;
--
    FUNCTION VitalNOR4   (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
```

Copyright © 2001 IEEE. All rights reserved.

```
        RETURN ResultMap(NOT (a OR b OR c OR d));
    END;
--
    FUNCTION VitalXNOR4  (
            CONSTANT a, b, c, d :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT (a XOR b XOR c XOR d));
    END;

    -- ---------------------------------------------------------------------------
    -- Buffers
    --   BUF    ....... standard non-inverting buffer
    --   BUFIF0 ....... non-inverting buffer Data passes thru if (Enable = '0')
    --   BUFIF1 ....... non-inverting buffer Data passes thru if (Enable = '1')
    -- ---------------------------------------------------------------------------
    FUNCTION VitalBUF    (
            CONSTANT         Data :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                     := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(To_UX01(Data));
    END;
--
    FUNCTION VitalBUFIF0 (
            CONSTANT Data, Enable :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                     := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(BufIf0_Table(Enable,Data));
    END;
--
    FUNCTION VitalBUFIF1 (
            CONSTANT Data, Enable :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                     := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(BufIf1_Table(Enable,Data));
    END;
    FUNCTION VitalIDENT  (
            CONSTANT         Data :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                     := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(To_UX01Z(Data));
    END;

    -- ---------------------------------------------------------------------------
    -- Invertors
    --   INV    ......... standard inverting buffer
    --   INVIF0 ......... inverting buffer Data passes thru if (Enable = '0')
    --   INVIF1 ......... inverting buffer Data passes thru if (Enable = '1')
    -- ---------------------------------------------------------------------------
    FUNCTION VitalINV    (
            CONSTANT         Data :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                     := VitalDefaultResultMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(NOT Data);
    END;
--
    FUNCTION VitalINVIF0 (
            CONSTANT Data, Enable :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                     := VitalDefaultResultZMap
          ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(InvIf0_Table(Enable,Data));
    END;
--
    FUNCTION VitalINVIF1 (
            CONSTANT Data, Enable :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                     := VitalDefaultResultZMap
```

 Copyright © 2001 IEEE. All rights reserved.

```
      ) RETURN std_ulogic IS
    BEGIN
        RETURN ResultMap(InvIf1_Table(Enable,Data));
    END;

    -- ------------------------------------------------------------------------
    -- Multiplexor
    --   MUX   .......... result := data(dselect)
    --   MUX2  .......... 2-input mux; result := data0 when (dselect = '0'),
    --                                           data1 when (dselect = '1'),
    --                         'X' when (dselect = 'X') and (data0 /= data1)
    --   MUX4  .......... 4-input mux; result := data(dselect)
    --   MUX8  .......... 8-input mux; result := data(dselect)
    -- ------------------------------------------------------------------------
    FUNCTION VitalMUX2  (
            CONSTANT Data1, Data0 :  IN std_ulogic;
            CONSTANT      dSelect :  IN std_ulogic;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
      ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        CASE To_X01(dSelect) IS
          WHEN '0'     => Result := To_UX01(Data0);
          WHEN '1'     => Result := To_UX01(Data1);
          WHEN OTHERS => Result := VitalSame( Data1, Data0 );
        END CASE;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalMUX4  (
            CONSTANT       Data :  IN std_logic_vector4;
            CONSTANT    dSelect :  IN std_logic_vector2;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
      ) RETURN std_ulogic IS
        VARIABLE Slct : std_logic_vector2;
        VARIABLE Result : UX01;
    BEGIN
        Slct := To_X01(dSelect);
        CASE Slct IS
          WHEN "00"   => Result := To_UX01(Data(0));
          WHEN "01"   => Result := To_UX01(Data(1));
          WHEN "10"   => Result := To_UX01(Data(2));
          WHEN "11"   => Result := To_UX01(Data(3));
          WHEN "0X"   => Result := VitalSame( Data(1), Data(0) );
          WHEN "1X"   => Result := VitalSame( Data(2), Data(3) );
          WHEN "X0"   => Result := VitalSame( Data(2), Data(0) );
          WHEN "X1"   => Result := VitalSame( Data(3), Data(1) );
          WHEN OTHERS => Result := VitalSame( VitalSame(Data(3),Data(2)),
                                              VitalSame(Data(1),Data(0)));
        END CASE;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VitalMUX8  (
            CONSTANT       Data :  IN std_logic_vector8;
            CONSTANT    dSelect :  IN std_logic_vector3;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
      ) RETURN std_ulogic IS
        VARIABLE Result : UX01;
    BEGIN
        CASE To_X01(dSelect(2)) IS
          WHEN '0'     => Result := VitalMUX4( Data(3 DOWNTO 0),
                                              dSelect(1 DOWNTO 0));
          WHEN '1'     => Result := VitalMUX4( Data(7 DOWNTO 4),
                                              dSelect(1 DOWNTO 0));
          WHEN OTHERS => Result := VitalSame( VitalMUX4( Data(3 DOWNTO 0),
                                                         dSelect(1 DOWNTO 0)),
                                              VitalMUX4( Data(7 DOWNTO 4),
                                                         dSelect(1 DOWNTO 0)));
        END CASE;
        RETURN ResultMap(Result);
    END;
--
    FUNCTION VInterMux    (
            CONSTANT Data    : IN std_logic_vector;
            CONSTANT dSelect : IN std_logic_vector
      ) RETURN std_ulogic IS
```

Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT sMsb     : INTEGER := dSelect'LENGTH;
        CONSTANT dMsbHigh : INTEGER := Data'LENGTH;
        CONSTANT dMsbLow  : INTEGER := Data'LENGTH/2;
        ALIAS DataAlias : std_logic_vector (   Data'LENGTH DOWNTO 1) IS Data;
        ALIAS dSelAlias : std_logic_vector (dSelect'LENGTH DOWNTO 1) IS dSelect;

        VARIABLE Result : UX01;
    BEGIN
        IF dSelect'LENGTH = 1 THEN
            Result := VitalMUX2( DataAlias(2), DataAlias(1), dSelAlias(1) );
        ELSIF dSelect'LENGTH = 2 THEN
            Result := VitalMUX4( DataAlias, dSelAlias );
        ELSIF dSelect'LENGTH > 2 THEN
          CASE To_X01(dSelect(sMsb)) IS
            WHEN '0'     =>
              Result := VInterMux( DataAlias(dMsbLow  DOWNTO          1),
                                   dSelAlias(sMsb-1 DOWNTO 1) );
            WHEN '1'     =>
              Result := VInterMux( DataAlias(dMsbHigh DOWNTO dMsbLow+1),
                                   dSelAlias(sMsb-1 DOWNTO 1) );
            WHEN OTHERS =>
              Result := VitalSame(
                              VInterMux( DataAlias(dMsbLow  DOWNTO          1),
                                         dSelAlias(sMsb-1 DOWNTO 1) ),
                              VInterMux( DataAlias(dMsbHigh DOWNTO dMsbLow+1),
                                         dSelAlias(sMsb-1 DOWNTO 1) )
                              );
          END CASE;
        ELSE
          Result := 'X'; -- dselect'LENGTH < 1
        END IF;
        RETURN Result;
    END;
--
    FUNCTION VitalMUX   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT    dSelect :  IN std_logic_vector;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_ulogic IS
        CONSTANT msb    : INTEGER := 2**dSelect'LENGTH;
        ALIAS DataAlias : std_logic_vector (   Data'LENGTH DOWNTO 1) IS Data;
        ALIAS dSelAlias : std_logic_vector (dSelect'LENGTH DOWNTO 1) IS dSelect;
        VARIABLE lDat   : std_logic_vector(msb DOWNTO 1) := (OTHERS=>'X');
        VARIABLE Result : UX01;
    BEGIN
        IF Data'LENGTH <= msb THEN
            FOR i IN Data'LENGTH DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
        ELSE
            FOR i IN msb DOWNTO 1 LOOP
                lDat(i) := DataAlias(i);
            END LOOP;
        END IF;
        Result := VInterMux( lDat, dSelAlias );
        RETURN ResultMap(Result);
    END;

    -- ------------------------------------------------------------------------
    -- Decoder
    --          General Algorithm :
    --              (a) Result(...) := '0' when (enable = '0')
    --              (b) Result(data) := '1'; all other subelements = '0'
    --              ... Result array is decending (n-1 downto 0)
    --
    --          DECODERn  .......... n:2**n decoder
    -- ------------------------------------------------------------------------
    FUNCTION VitalDECODER2  (
            CONSTANT       Data :  IN std_ulogic;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
          ) RETURN std_logic_vector2 IS
        VARIABLE Result : std_logic_vector2;
    BEGIN
        Result(1) := ResultMap(Enable AND (    Data));
        Result(0) := ResultMap(Enable AND (NOT Data));
        RETURN Result;
    END;
--
```

 Copyright © 2001 IEEE. All rights reserved.

```
    FUNCTION VitalDECODER4  (
            CONSTANT       Data :  IN std_logic_vector2;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
        ) RETURN std_logic_vector4 IS
        VARIABLE Result : std_logic_vector4;
    BEGIN
        Result(3) := ResultMap(Enable AND (    Data(1)) AND (    Data(0)));
        Result(2) := ResultMap(Enable AND (    Data(1)) AND (NOT Data(0)));
        Result(1) := ResultMap(Enable AND (NOT Data(1)) AND (    Data(0)));
        Result(0) := ResultMap(Enable AND (NOT Data(1)) AND (NOT Data(0)));
        RETURN Result;
    END;
--
    FUNCTION VitalDECODER8  (
            CONSTANT       Data :  IN std_logic_vector3;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
        ) RETURN std_logic_vector8 IS
        VARIABLE Result : std_logic_vector8;
    BEGIN
        Result(7) := (    Data(2)) AND (    Data(1)) AND (    Data(0));
        Result(6) := (    Data(2)) AND (    Data(1)) AND (NOT Data(0));
        Result(5) := (    Data(2)) AND (NOT Data(1)) AND (    Data(0));
        Result(4) := (    Data(2)) AND (NOT Data(1)) AND (NOT Data(0));
        Result(3) := (NOT Data(2)) AND (    Data(1)) AND (    Data(0));
        Result(2) := (NOT Data(2)) AND (    Data(1)) AND (NOT Data(0));
        Result(1) := (NOT Data(2)) AND (NOT Data(1)) AND (    Data(0));
        Result(0) := (NOT Data(2)) AND (NOT Data(1)) AND (NOT Data(0));

        Result(0) := ResultMap ( Enable AND Result(0) );
        Result(1) := ResultMap ( Enable AND Result(1) );
        Result(2) := ResultMap ( Enable AND Result(2) );
        Result(3) := ResultMap ( Enable AND Result(3) );
        Result(4) := ResultMap ( Enable AND Result(4) );
        Result(5) := ResultMap ( Enable AND Result(5) );
        Result(6) := ResultMap ( Enable AND Result(6) );
        Result(7) := ResultMap ( Enable AND Result(7) );

        RETURN Result;
    END;
--
    FUNCTION VitalDECODER   (
            CONSTANT       Data :  IN std_logic_vector;
            CONSTANT     Enable :  IN std_ulogic;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
        ) RETURN std_logic_vector IS

        CONSTANT DMsb : INTEGER := Data'LENGTH - 1;
        ALIAS DataAlias : std_logic_vector ( DMsb DOWNTO 0 ) IS Data;
    BEGIN
        IF    Data'LENGTH = 1 THEN
            RETURN VitalDECODER2 (DataAlias(    0     ), Enable, ResultMap );
        ELSIF Data'LENGTH = 2 THEN
            RETURN VitalDECODER4 (DataAlias(1 DOWNTO 0), Enable, ResultMap );
        ELSIF Data'LENGTH = 3 THEN
            RETURN VitalDECODER8 (DataAlias(2 DOWNTO 0), Enable, ResultMap );
        ELSIF Data'LENGTH > 3 THEN
            RETURN VitalDECODER  (DataAlias(DMsb-1 DOWNTO 0),
                                   Enable AND (    DataAlias(DMsb)), ResultMap )
                 & VitalDECODER  (DataAlias(DMsb-1 DOWNTO 0),
                                  Enable AND (NOT DataAlias(DMsb)), ResultMap );
        ELSE RETURN "X";
        END IF;
    END;

    -- ------------------------------------------------------------------------
    -- N-bit wide Logical gates.
    -- ------------------------------------------------------------------------
    PROCEDURE VitalAND   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
```

```
        VARIABLE Data_Schd : SchedArray(Data'RANGE);
        VARIABLE NewValue     : UX01;
        VARIABLE Glitch_Data : GlitchDataType;
        VARIABLE new_schd    : SchedType;
        VARIABLE Dly, Glch   : TIME;
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalAND(Data, ResultMap);
          WAIT ON Data;
      END LOOP;
      ELSE

        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        FOR n IN Data'RANGE LOOP
            BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := '1';
        new_schd := Data_Schd(Data_Schd'LEFT);
        FOR i IN Data'RANGE LOOP
            NewValue  := NewValue  AND Data(i);
            new_schd := new_schd AND Data_Schd(i);
        END LOOP;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data;
      END LOOP;
      END IF; --SN
    END;
--
    PROCEDURE VitalOR    (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
        VARIABLE Data_Schd : SchedArray(Data'RANGE);
        VARIABLE NewValue     : UX01;
        VARIABLE Glitch_Data : GlitchDataType;
        VARIABLE new_schd    : SchedType;
        VARIABLE Dly, Glch   : TIME;
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------------------------------------------
        --  Check if ALL zero delay paths, use simple model
        --    ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        FOR i IN Data'RANGE LOOP
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;
        IF (AllZeroDelay) THEN LOOP
            q <= VitalOR(Data, ResultMap);
            WAIT ON Data;
        END LOOP;
        ELSE

          -- --------------------------------------
          -- Initialize delay schedules
          -- --------------------------------------
          FOR n IN Data'RANGE LOOP
              BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
          END LOOP;

        LOOP
          -- --------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- --------------------------------------
          GetEdge ( Data, LastData, Data_Edge );
          BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

          -- ------------------------------------
          -- Compute function and propation delay
          -- ------------------------------------
          NewValue := '0';
          new_schd := Data_Schd(Data_Schd'LEFT);
          FOR i IN Data'RANGE LOOP
              NewValue  := NewValue  OR Data(i);
              new_schd := new_schd OR Data_Schd(i);
          END LOOP;

          -- ------------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ------------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                               PrimGlitchMode, GlitchDelay=>Glch );

          WAIT ON Data;
        END LOOP;
        END IF; --SN
      END;
--
      PROCEDURE VitalXOR   (
              SIGNAL            q : OUT std_ulogic;
              SIGNAL         Data :  IN std_logic_vector;
              CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
              CONSTANT  ResultMap :  IN VitalResultMapType
                                          := VitalDefaultResultMap
      ) IS
          VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
          VARIABLE Data_Edge   :  EdgeArray(Data'RANGE);
          VARIABLE DataB_Schd  : SchedArray(1 TO Data'LENGTH);
          VARIABLE DataI_Schd  : SchedArray(1 TO Data'LENGTH);
          VARIABLE NewValue     : UX01;
          VARIABLE Glitch_Data : GlitchDataType;
          VARIABLE new_schd    : SchedType;
          VARIABLE Dly, Glch   : TIME;
          ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
          ALIAS ADataB_Schd : SchedArray(Data'RANGE) IS DataB_Schd;
          ALIAS ADataI_Schd : SchedArray(Data'RANGE) IS DataI_Schd;
          VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
      BEGIN
        -- ------------------------------------------------------------------------
        --  Check if ALL zero delay paths, use simple model
        --    ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        FOR i IN Data'RANGE LOOP
```

```
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;
        IF (AllZeroDelay) THEN LOOP
            q <= VitalXOR(Data, ResultMap);
            WAIT ON Data;
        END LOOP;
        ELSE

          -- --------------------------------------
          -- Initialize delay schedules
          -- --------------------------------------
          FOR n IN Data'RANGE LOOP
              BufPath ( ADataB_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
              InvPath ( ADataI_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
          END LOOP;

        LOOP
          -- --------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- --------------------------------------
          GetEdge ( Data, LastData, Data_Edge );
          BufPath ( ADataB_Schd, Data_Edge, Atpd_data_q );
          InvPath ( ADataI_Schd, Data_Edge, Atpd_data_q );

          -- ------------------------------------
          -- Compute function and propation delay
          -- ------------------------------------
          NewValue  := VitalXOR ( Data );
          new_schd := VitalXOR ( DataB_Schd, DataI_Schd );

          -- ------------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ------------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                               PrimGlitchMode, GlitchDelay=>Glch );

          WAIT ON Data;
        END LOOP;
        END IF; --SN
      END;
--
    PROCEDURE VitalNAND  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
          VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
          VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
          VARIABLE Data_Schd : SchedArray(Data'RANGE);
          VARIABLE NewValue     : UX01;
          VARIABLE Glitch_Data : GlitchDataType;
          VARIABLE new_schd    : SchedType;
          VARIABLE Dly, Glch   : TIME;
          ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
          VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
        -- ------------------------------------------------------------------------
        --  Check if ALL zero delay paths, use simple model
        --   ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        FOR i IN Data'RANGE LOOP
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;
        IF (AllZeroDelay) THEN LOOP
            q <= VitalNAND(Data, ResultMap);
            WAIT ON Data;
        END LOOP;
        ELSE
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- -------------------------------------
        -- Initialize delay schedules
        -- -------------------------------------
        FOR n IN Data'RANGE LOOP
            InvPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;

      LOOP
        -- -------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- -------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        InvPath ( Data_Schd, Data_Edge, Atpd_data_q );

        -- -----------------------------------
        -- Compute function and propation delay
        -- -----------------------------------
        NewValue := '1';
        new_schd := Data_Schd(Data_Schd'LEFT);
        FOR i IN Data'RANGE LOOP
            NewValue  := NewValue  AND Data(i);
            new_schd := new_schd AND Data_Schd(i);
        END LOOP;
        NewValue  := NOT NewValue;
        new_schd := NOT new_schd;

        -- ---------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ---------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data;
      END LOOP;
      END IF;
    END;
--
    PROCEDURE VitalNOR   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE Data_Edge :  EdgeArray(Data'RANGE);
        VARIABLE Data_Schd : SchedArray(Data'RANGE);
        VARIABLE NewValue     : UX01;
        VARIABLE Glitch_Data : GlitchDataType;
        VARIABLE new_schd    : SchedType;
        VARIABLE Dly, Glch   : TIME;
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalNOR(Data, ResultMap);
          WAIT ON Data;
      END LOOP;
      ELSE

        -- -------------------------------------
        -- Initialize delay schedules
        -- -------------------------------------
        FOR n IN Data'RANGE LOOP
            InvPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
```

Copyright © 2001 IEEE. All rights reserved.

```
        END LOOP;

      LOOP
        -- -------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- -------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        InvPath ( Data_Schd, Data_Edge, Atpd_data_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := '0';
        new_schd := Data_Schd(Data_Schd'LEFT);
        FOR i IN Data'RANGE LOOP
            NewValue  := NewValue  OR Data(i);
            new_schd := new_schd OR Data_Schd(i);
        END LOOP;
        NewValue  := NOT NewValue;
        new_schd := NOT new_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data;
      END LOOP;
      END IF; --SN
    END;
--
    PROCEDURE VitalXNOR  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE LastData    : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE Data_Edge   :  EdgeArray(Data'RANGE);
        VARIABLE DataB_Schd  : SchedArray(1 TO Data'LENGTH);
        VARIABLE DataI_Schd  : SchedArray(1 TO Data'LENGTH);
        VARIABLE NewValue     : UX01;
        VARIABLE Glitch_Data : GlitchDataType;
        VARIABLE new_schd    : SchedType;
        VARIABLE Dly, Glch   : TIME;
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        ALIAS ADataB_Schd : SchedArray(Data'RANGE) IS DataB_Schd;
        ALIAS ADataI_Schd : SchedArray(Data'RANGE) IS DataI_Schd;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalXNOR(Data, ResultMap);
          WAIT ON Data;
      END LOOP;
      ELSE

        -- -------------------------------------
        -- Initialize delay schedules
        -- -------------------------------------
        FOR n IN Data'RANGE LOOP
            BufPath ( ADataB_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
            InvPath ( ADataI_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;
```


Copyright © 2001 IEEE. All rights reserved.

```
        LOOP
          -- --------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- --------------------------------------
          GetEdge ( Data, LastData, Data_Edge );
          BufPath ( ADataB_Schd, Data_Edge, Atpd_data_q );
          InvPath ( ADataI_Schd, Data_Edge, Atpd_data_q );

          -- ------------------------------------
          -- Compute function and propation delay
          -- ------------------------------------
          NewValue  := VitalXNOR ( Data );
          new_schd := VitalXNOR ( DataB_Schd, DataI_Schd );

          -- ------------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ------------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                               PrimGlitchMode, GlitchDelay=>Glch );

          WAIT ON Data;
        END LOOP;
        END IF; --SN
      END;
  --

      -- ------------------------------------------------------------------------
      -- Commonly used 2-bit Logical gates.
      -- ------------------------------------------------------------------------
      PROCEDURE VitalAND2  (
              SIGNAL            q : OUT std_ulogic;
              SIGNAL         a, b :  IN std_ulogic   ;
              CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
              CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
              CONSTANT  ResultMap :  IN VitalResultMapType
                                         := VitalDefaultResultMap
      ) IS
          VARIABLE a_schd, b_schd : SchedType;
          VARIABLE NewValue        : UX01;
          VARIABLE Glitch_Data    : GlitchDataType;
          VARIABLE new_schd       : SchedType;
          VARIABLE Dly, Glch      : TIME;
      BEGIN

      -- ------------------------------------------------------------------------
      --  For ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
        LOOP
          q <= VitalAND2 ( a, b, ResultMap );
          WAIT ON a, b;
        END LOOP;

      ELSE
          -- --------------------------------------
          -- Initialize delay schedules
          -- --------------------------------------
          BufPath ( a_schd, InitialEdge(a), tpd_a_q );
          BufPath ( b_schd, InitialEdge(b), tpd_b_q );

        LOOP
          -- --------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- --------------------------------------
          BufPath ( a_schd, GetEdge(a), tpd_a_q );
          BufPath ( b_schd, GetEdge(b), tpd_b_q );

          -- ------------------------------------
          -- Compute function and propation delay
          -- ------------------------------------
          NewValue := a AND b;
          new_schd := a_schd AND b_schd;
```

Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalOR2   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalOR2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := a OR b;
        new_schd := a_schd OR b_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNAND2 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
```

 Copyright © 2001 IEEE. All rights reserved.

```
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNAND2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := a NAND b;
        new_schd := a_schd NAND b_schd;

        -- ----------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ----------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNOR2  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNOR2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
```

Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------
        -- Initialize delay schedules
        -- ------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- ------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );

        -- ----------------------------------
        -- Compute function and propation delay
        -- ----------------------------------
        NewValue := a NOR b;
        new_schd := a_schd NOR b_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXOR2  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd : SchedType;
        VARIABLE ai_schd, bi_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data     : GlitchDataType;
        VARIABLE new_schd        : SchedType;
        VARIABLE Dly, Glch       : TIME;
    BEGIN

    -- ----------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ----------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXOR2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- ------------------------------------
        -- Initialize delay schedules
        -- ------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- ------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ------------------------------------
        BufPath ( ab_schd, GetEdge(a), tpd_a_q );
        InvPath ( ai_schd, GetEdge(a), tpd_a_q );

        BufPath ( bb_schd, GetEdge(b), tpd_b_q );
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a XOR b;
        new_schd := VitalXOR2 ( ab_schd,ai_schd, bb_schd,bi_schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXNOR2 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         a, b :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd : SchedType;
        VARIABLE ai_schd, bi_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data     : GlitchDataType;
        VARIABLE new_schd        : SchedType;
        VARIABLE Dly, Glch       : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF ((tpd_a_q = VitalZeroDelay01) AND (tpd_b_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXNOR2 ( a, b, ResultMap );
        WAIT ON a, b;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( ab_schd, GetEdge(a), tpd_a_q );
        InvPath ( ai_schd, GetEdge(a), tpd_a_q );

        BufPath ( bb_schd, GetEdge(b), tpd_b_q );
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := NOT (a XOR b);
        new_schd := VitalXNOR2 ( ab_schd,ai_schd, bb_schd,bi_schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
```

Copyright © 2001 IEEE. All rights reserved.

```
                         PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b;
      END LOOP;
    END IF;
    END;

    -- ------------------------------------------------------------------------
    -- Commonly used 3-bit Logical gates.
    -- ------------------------------------------------------------------------
    PROCEDURE VitalAND3  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                     := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN
--
    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalAND3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );
        BufPath ( c_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );
        BufPath ( c_schd, GetEdge(c), tpd_c_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a AND b AND c;
        new_schd := a_schd AND b_schd AND c_schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalOR3   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
) IS
    VARIABLE a_schd, b_schd, c_schd : SchedType;
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--    ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF (     (tpd_a_q = VitalZeroDelay01)
     AND (tpd_b_q = VitalZeroDelay01)
     AND (tpd_c_q = VitalZeroDelay01)) THEN
  LOOP
    q <= VitalOR3 ( a, b, c, ResultMap );
    WAIT ON a, b, c;
  END LOOP;

ELSE
    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    BufPath ( a_schd, InitialEdge(a), tpd_a_q );
    BufPath ( b_schd, InitialEdge(b), tpd_b_q );
    BufPath ( c_schd, InitialEdge(c), tpd_c_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    BufPath ( a_schd, GetEdge(a), tpd_a_q );
    BufPath ( b_schd, GetEdge(b), tpd_b_q );
    BufPath ( c_schd, GetEdge(c), tpd_c_q );

    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := a OR b OR c;
    new_schd := a_schd OR b_schd OR c_schd;

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, “q”, Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON a, b, c;
  END LOOP;
END IF;
END;
--
PROCEDURE VitalNAND3 (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL      a, b, c :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                := VitalDefaultResultMap
) IS
    VARIABLE a_schd, b_schd, c_schd : SchedType;
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--    ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
```

Copyright © 2001 IEEE. All rights reserved.

```
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNAND3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- -------------------------------------
        -- Initialize delay schedules
        -- -------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- -------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- -------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );
        InvPath ( c_schd, GetEdge(c), tpd_c_q );

        -- -----------------------------------
        -- Compute function and propation delay
        -- -----------------------------------
        NewValue  := (a AND b) NAND c;
        new_schd := (a_schd AND b_schd) NAND c_schd;

        -- ----------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ----------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalNOR3  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ---------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNOR3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- -------------------------------------
        -- Initialize delay schedules
        -- -------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
          LOOP
            -- --------------------------------------
            -- Process input signals
            --   get edge values
            --   re-evaluate output schedules
            -- --------------------------------------
            InvPath ( a_schd, GetEdge(a), tpd_a_q );
            InvPath ( b_schd, GetEdge(b), tpd_b_q );
            InvPath ( c_schd, GetEdge(c), tpd_c_q );

            -- ------------------------------------
            -- Compute function and propation delay
            -- ------------------------------------
            NewValue  := (a OR b) NOR c;
            new_schd := (a_schd OR b_schd) NOR c_schd;

            -- ------------------------------------------------------
            -- Assign Outputs
            --  get delays to new value and possable glitch
            --  schedule output change with On Event glitch detection
            -- ------------------------------------------------------
            GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
            VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                                 PrimGlitchMode, GlitchDelay=>Glch );

            WAIT ON a, b, c;
          END LOOP;
        END IF;
        END;
    --
        PROCEDURE VitalXOR3  (
                SIGNAL            q : OUT std_ulogic;
                SIGNAL      a, b, c :  IN std_ulogic   ;
                CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
                CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
        ) IS
            VARIABLE ab_schd, bb_schd, cb_schd : SchedType;
            VARIABLE ai_schd, bi_schd, ci_schd : SchedType;
            VARIABLE NewValue        : UX01;
            VARIABLE Glitch_Data    : GlitchDataType;
            VARIABLE new_schd       : SchedType;
            VARIABLE Dly, Glch      : TIME;
        BEGIN

        -- ------------------------------------------------------------------------
        --  For ALL zero delay paths, use simple model
        --   ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        IF (     (tpd_a_q = VitalZeroDelay01)
             AND (tpd_b_q = VitalZeroDelay01)
             AND (tpd_c_q = VitalZeroDelay01)) THEN
          LOOP
            q <= VitalXOR3 ( a, b, c, ResultMap );
            WAIT ON a, b, c;
          END LOOP;

        ELSE
            -- --------------------------------------
            -- Initialize delay schedules
            -- --------------------------------------
            BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
            InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
            BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
            InvPath ( bi_schd, InitialEdge(b), tpd_b_q );
            BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
            InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

          LOOP
            -- --------------------------------------
            -- Process input signals
            --   get edge values
            --   re-evaluate output schedules
            -- --------------------------------------
            BufPath ( ab_schd, GetEdge(a), tpd_a_q );
            InvPath ( ai_schd, GetEdge(a), tpd_a_q );

            BufPath ( bb_schd, GetEdge(b), tpd_b_q );
```

Copyright © 2001 IEEE. All rights reserved.

```
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );

        BufPath ( cb_schd, GetEdge(c), tpd_c_q );
        InvPath ( ci_schd, GetEdge(c), tpd_c_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a XOR b XOR c;
        new_schd := VitalXOR3 ( ab_schd,ai_schd,
                                bb_schd,bi_schd,
                                cb_schd,ci_schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXNOR3 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL      a, b, c :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd, cb_schd : SchedType;
        VARIABLE ai_schd, bi_schd, ci_schd : SchedType;
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXNOR3 ( a, b, c, ResultMap );
        WAIT ON a, b, c;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );
        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );
        BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
        InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( ab_schd, GetEdge(a), tpd_a_q );
        InvPath ( ai_schd, GetEdge(a), tpd_a_q );

        BufPath ( bb_schd, GetEdge(b), tpd_b_q );
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );

        BufPath ( cb_schd, GetEdge(c), tpd_c_q );
        InvPath ( ci_schd, GetEdge(c), tpd_c_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := NOT (a XOR b XOR c);
    new_schd := VitalXNOR3 ( ab_schd, ai_schd,
                             bb_schd, bi_schd,
                             cb_schd, ci_schd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON a, b, c;
  END LOOP;
END IF;
END;

-- ------------------------------------------------------------------------
-- Commonly used 4-bit Logical gates.
-- ------------------------------------------------------------------------
PROCEDURE VitalAND4  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
) IS
    VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF (     (tpd_a_q = VitalZeroDelay01)
     AND (tpd_b_q = VitalZeroDelay01)
     AND (tpd_c_q = VitalZeroDelay01)
     AND (tpd_d_q = VitalZeroDelay01)) THEN
  LOOP
    q <= VitalAND4 ( a, b, c, d, ResultMap );
    WAIT ON a, b, c, d;
  END LOOP;

ELSE
    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    BufPath ( a_schd, InitialEdge(a), tpd_a_q );
    BufPath ( b_schd, InitialEdge(b), tpd_b_q );
    BufPath ( c_schd, InitialEdge(c), tpd_c_q );
    BufPath ( d_Schd, InitialEdge(d), tpd_d_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    BufPath ( a_schd, GetEdge(a), tpd_a_q );
    BufPath ( b_schd, GetEdge(b), tpd_b_q );
    BufPath ( c_schd, GetEdge(c), tpd_c_q );
    BufPath ( d_Schd, GetEdge(d), tpd_d_q );

    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := a AND b AND c AND d;
    new_schd := a_schd AND b_schd AND c_schd AND d_Schd;
```

Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalOR4   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
        VARIABLE NewValue       : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalOR4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( a_schd, InitialEdge(a), tpd_a_q );
        BufPath ( b_schd, InitialEdge(b), tpd_b_q );
        BufPath ( c_schd, InitialEdge(c), tpd_c_q );
        BufPath ( d_Schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( a_schd, GetEdge(a), tpd_a_q );
        BufPath ( b_schd, GetEdge(b), tpd_b_q );
        BufPath ( c_schd, GetEdge(c), tpd_c_q );
        BufPath ( d_Schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a OR b OR c OR d;
        new_schd := a_schd OR b_schd OR c_schd OR d_Schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        END LOOP;
      END IF;
      END;
  --
    PROCEDURE VitalNAND4 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                     := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
        VARIABLE NewValue         : UX01;
        VARIABLE Glitch_Data      : GlitchDataType;
        VARIABLE new_schd         : SchedType;
        VARIABLE Dly, Glch        : TIME;
    BEGIN

    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNAND4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- ---------------------------------------
        -- Initialize delay schedules
        -- ---------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );
        InvPath ( d_Schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- ---------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ---------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );
        InvPath ( c_schd, GetEdge(c), tpd_c_q );
        InvPath ( d_Schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := (a AND b) NAND (c AND d);
        new_schd := (a_schd AND b_schd) NAND (c_schd AND d_Schd);

        -- ---------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ---------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
  --
    PROCEDURE VitalNOR4  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
```

Copyright © 2001 IEEE. All rights reserved.

```
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
    ) IS
        VARIABLE a_schd, b_schd, c_schd, d_Schd : SchedType;
        VARIABLE NewValue         : UX01;
        VARIABLE Glitch_Data     : GlitchDataType;
        VARIABLE new_schd        : SchedType;
        VARIABLE Dly, Glch       : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalNOR4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        InvPath ( a_schd, InitialEdge(a), tpd_a_q );
        InvPath ( b_schd, InitialEdge(b), tpd_b_q );
        InvPath ( c_schd, InitialEdge(c), tpd_c_q );
        InvPath ( d_Schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        InvPath ( a_schd, GetEdge(a), tpd_a_q );
        InvPath ( b_schd, GetEdge(b), tpd_b_q );
        InvPath ( c_schd, GetEdge(c), tpd_c_q );
        InvPath ( d_Schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := (a OR b) NOR (c OR d);
        new_schd := (a_schd OR b_schd) NOR (c_schd OR d_Schd);

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXOR4  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd, cb_schd, DB_Schd : SchedType;
        VARIABLE ai_schd, bi_schd, ci_schd, di_schd : SchedType;
        VARIABLE NewValue         : UX01;
        VARIABLE Glitch_Data     : GlitchDataType;
        VARIABLE new_schd        : SchedType;
        VARIABLE Dly, Glch       : TIME;
    BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
    -- ---------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------------
    IF (     (tpd_a_q = VitalZeroDelay01)
         AND (tpd_b_q = VitalZeroDelay01)
         AND (tpd_c_q = VitalZeroDelay01)
         AND (tpd_d_q = VitalZeroDelay01)) THEN
      LOOP
        q <= VitalXOR4 ( a, b, c, d, ResultMap );
        WAIT ON a, b, c, d;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
        InvPath ( ai_schd, InitialEdge(a), tpd_a_q );

        BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
        InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

        BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
        InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

        BufPath ( DB_Schd, InitialEdge(d), tpd_d_q );
        InvPath ( di_schd, InitialEdge(d), tpd_d_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( ab_schd, GetEdge(a), tpd_a_q );
        InvPath ( ai_schd, GetEdge(a), tpd_a_q );

        BufPath ( bb_schd, GetEdge(b), tpd_b_q );
        InvPath ( bi_schd, GetEdge(b), tpd_b_q );

        BufPath ( cb_schd, GetEdge(c), tpd_c_q );
        InvPath ( ci_schd, GetEdge(c), tpd_c_q );

        BufPath ( DB_Schd, GetEdge(d), tpd_d_q );
        InvPath ( di_schd, GetEdge(d), tpd_d_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := a XOR b XOR c XOR d;
        new_schd := VitalXOR4 ( ab_schd,ai_schd, bb_schd,bi_schd,
                                cb_schd,ci_schd, DB_Schd,di_schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON a, b, c, d;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalXNOR4 (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL   a, b, c, d :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                      := VitalDefaultResultMap
    ) IS
        VARIABLE ab_schd, bb_schd, cb_schd, DB_Schd : SchedType;
        VARIABLE ai_schd, bi_schd, ci_schd, di_schd : SchedType;
        VARIABLE NewValue        : UX01;
```

Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
BEGIN

-- ---------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ---------------------------------------------------------------------------
IF (     (tpd_a_q = VitalZeroDelay01)
     AND (tpd_b_q = VitalZeroDelay01)
     AND (tpd_c_q = VitalZeroDelay01)
     AND (tpd_d_q = VitalZeroDelay01)) THEN
  LOOP
    q <= VitalXNOR4 ( a, b, c, d, ResultMap );
    WAIT ON a, b, c, d;
  END LOOP;

ELSE
    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    BufPath ( ab_schd, InitialEdge(a), tpd_a_q );
    InvPath ( ai_schd, InitialEdge(a), tpd_a_q );

    BufPath ( bb_schd, InitialEdge(b), tpd_b_q );
    InvPath ( bi_schd, InitialEdge(b), tpd_b_q );

    BufPath ( cb_schd, InitialEdge(c), tpd_c_q );
    InvPath ( ci_schd, InitialEdge(c), tpd_c_q );

    BufPath ( DB_Schd, InitialEdge(d), tpd_d_q );
    InvPath ( di_schd, InitialEdge(d), tpd_d_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    BufPath ( ab_schd, GetEdge(a), tpd_a_q );
    InvPath ( ai_schd, GetEdge(a), tpd_a_q );

    BufPath ( bb_schd, GetEdge(b), tpd_b_q );
    InvPath ( bi_schd, GetEdge(b), tpd_b_q );

    BufPath ( cb_schd, GetEdge(c), tpd_c_q );
    InvPath ( ci_schd, GetEdge(c), tpd_c_q );

    BufPath ( DB_Schd, GetEdge(d), tpd_d_q );
    InvPath ( di_schd, GetEdge(d), tpd_d_q );

    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    NewValue  := NOT (a XOR b XOR c XOR d);
    new_schd := VitalXNOR4 ( ab_schd,ai_schd, bb_schd,bi_schd,
                             cb_schd,ci_schd, DB_Schd,di_schd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON a, b, c, d;
  END LOOP;
END IF;
END;

-- ---------------------------------------------------------------------------
-- Buffers
--   BUF    ....... standard non-inverting buffer
--   BUFIF0 ....... non-inverting buffer Data passes thru if (Enable = '0')
--   BUFIF1 ....... non-inverting buffer Data passes thru if (Enable = '1')
-- ---------------------------------------------------------------------------
PROCEDURE VitalBUF   (
        SIGNAL            q : OUT std_ulogic;
```

 Copyright © 2001 IEEE. All rights reserved.

```
            SIGNAL            a :  IN std_ulogic   ;
            CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap
    ) IS
        VARIABLE NewValue      : UX01;
        VARIABLE Glitch_Data  : GlitchDataType;
        VARIABLE Dly, Glch    : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (tpd_a_q = VitalZeroDelay01) THEN
      LOOP
        q <= ResultMap(To_UX01(a));
        WAIT ON a;
      END LOOP;

    ELSE
      LOOP
        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue  := To_UX01(a);     -- convert to forcing strengths
        CASE EdgeType'(GetEdge(a)) IS
          WHEN '1'|'/'|'R'|'r'  => Dly := tpd_a_q(tr01);
          WHEN '0'|'\'|'F'|'f'  => Dly := tpd_a_q(tr10);
          WHEN OTHERS           => Dly := Minimum (tpd_a_q(tr01), tpd_a_q(tr10));
        END CASE;

        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode );

        WAIT ON a;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalBUFIF1 (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
    ) IS
        VARIABLE NewValue        : UX01Z;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
        VARIABLE Dly, Glch      : TIME;
    BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_data_q   = VitalZeroDelay01 )
         AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
      LOOP
        q <= VitalBUFIF1( Data, Enable, ResultMap );
        WAIT ON Data, Enable;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        BufPath ( d_Schd, InitialEdge(Data), tpd_data_q );
        BufEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        BufPath ( d_Schd, GetEdge(Data), tpd_data_q );
        BufEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );
```

Copyright © 2001 IEEE. All rights reserved.

```
        -- -----------------------------------
        -- Compute function and propation delay
        -- -----------------------------------
        NewValue := VitalBUFIF1( Data, Enable );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                        d_Schd, e1_Schd, e0_Schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalBUFIF0 (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                       := VitalDefaultResultZMap
    ) IS
        VARIABLE NewValue        : UX01Z;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
        VARIABLE ne1_schd, ne0_schd : SchedType;
        VARIABLE Dly, Glch      : TIME;
  BEGIN

    -- ---------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ---------------------------------------------------------------------
    IF (     (tpd_data_q   = VitalZeroDelay01 )
         AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
      LOOP
        q <= VitalBUFIF0( Data, Enable, ResultMap );
        WAIT ON Data, Enable;
      END LOOP;

    ELSE
        -- -------------------------------------
        -- Initialize delay schedules
        -- -------------------------------------
        BufPath ( d_Schd, InitialEdge(Data), tpd_data_q );
        InvEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- -------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- -------------------------------------
        BufPath ( d_Schd, GetEdge(Data), tpd_data_q );
        InvEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );

        -- -----------------------------------
        -- Compute function and propation delay
        -- -----------------------------------
        NewValue := VitalBUFIF0( Data, Enable );
        ne1_schd := NOT e1_Schd;
        ne0_schd := NOT e0_Schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                        d_Schd, ne1_schd, ne0_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );
```

 Copyright © 2001 IEEE. All rights reserved.

```
    WAIT ON Data, Enable;
  END LOOP;
END IF;
END;

PROCEDURE VitalIDENT (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL            a :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
        CONSTANT  ResultMap :  IN VitalResultZMapType
                                   := VitalDefaultResultZMap
) IS
    SUBTYPE v2 IS std_logic_vector(0 TO 1);
    VARIABLE NewValue      : UX01Z;
    VARIABLE Glitch_Data  : GlitchDataType;
    VARIABLE Dly, Glch    : TIME;
BEGIN

-- ---------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--    ( No delay selection, glitch detection required )
-- ---------------------------------------------------------------------------
IF (tpd_a_q = VitalZeroDelay01Z) THEN
  LOOP
    q <= ResultMap(To_UX01Z(a));
    WAIT ON a;
  END LOOP;

ELSE
  LOOP
    -- ------------------------------------
    -- Compute function and propation delay
    -- ------------------------------------
    CASE v2'(To_X01Z(NewValue) & To_X01Z(a)) IS
      WHEN "00"    => Dly := tpd_a_q(tr10);
      WHEN "01"    => Dly := tpd_a_q(tr01);
      WHEN "0Z"    => Dly := tpd_a_q(tr0z);
      WHEN "0X"    => Dly := tpd_a_q(tr01);
      WHEN "10"    => Dly := tpd_a_q(tr10);
      WHEN "11"    => Dly := tpd_a_q(tr01);
      WHEN "1Z"    => Dly := tpd_a_q(tr1z);
      WHEN "1X"    => Dly := tpd_a_q(tr10);
      WHEN "Z0"    => Dly := tpd_a_q(trz0);
      WHEN "Z1"    => Dly := tpd_a_q(trz1);
      WHEN "ZZ"    => Dly := 0 ns;
      WHEN "ZX"    => Dly := Minimum (tpd_a_q(trz1), tpd_a_q(trz0));
      WHEN "X0"    => Dly := tpd_a_q(tr10);
      WHEN "X1"    => Dly := tpd_a_q(tr01);
      WHEN "XZ"    => Dly := Minimum (tpd_a_q(tr0z), tpd_a_q(tr1z));
      WHEN OTHERS => Dly := Minimum (tpd_a_q(tr01), tpd_a_q(tr10));
    END CASE;
    NewValue  := To_UX01Z(a);

    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode );

    WAIT ON a;
  END LOOP;
END IF;
END;

-- ---------------------------------------------------------------------------
-- Invertors
--   INV    ......... standard inverting buffer
--   INVIF0 ......... inverting buffer Data passes thru if (Enable = '0')
--   INVIF1 ......... inverting buffer Data passes thru if (Enable = '1')
-- ---------------------------------------------------------------------------
PROCEDURE VitalINV   (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL            a :  IN std_ulogic   ;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                   := VitalDefaultResultMap
) IS
    VARIABLE NewValue      : UX01;
    VARIABLE Glitch_Data  : GlitchDataType;
    VARIABLE new_schd     : SchedType;
    VARIABLE Dly, Glch    : TIME;
BEGIN
IF (tpd_a_q = VitalZeroDelay01) THEN
  LOOP
```

Copyright © 2001 IEEE. All rights reserved.

```
          q <= ResultMap(NOT a);
          WAIT ON a;
        END LOOP;

      ELSE
        LOOP
          -- ------------------------------------
          -- Compute function and propation delay
          -- ------------------------------------
          NewValue  := NOT a;
          CASE EdgeType' (GetEdge(a)) IS
            WHEN '1' |'/' |'R' |'r'  => Dly := tpd_a_q(tr10);
            WHEN '0' |'\' |'F' |'f'  => Dly := tpd_a_q(tr01);
            WHEN OTHERS           => Dly := Minimum (tpd_a_q(tr01), tpd_a_q(tr10));
          END CASE;

          VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                               PrimGlitchMode );

          WAIT ON a;
        END LOOP;
      END IF;
      END;
--
      PROCEDURE VitalINVIF1 (
              SIGNAL              q : OUT std_ulogic;
              SIGNAL           Data :  IN std_ulogic;
              SIGNAL         Enable :  IN std_ulogic;
              CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
              CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
              CONSTANT    ResultMap :  IN VitalResultZMapType
                                         := VitalDefaultResultZMap
      ) IS
          VARIABLE NewValue        : UX01Z;
          VARIABLE new_schd       : SchedType;
          VARIABLE Glitch_Data    : GlitchDataType;
          VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
          VARIABLE Dly, Glch      : TIME;
    BEGIN

      -- ---------------------------------------------------------------------
      --  For ALL zero delay paths, use simple model
      --   ( No delay selection, glitch detection required )
      -- ---------------------------------------------------------------------
      IF (     (tpd_data_q   = VitalZeroDelay01 )
           AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
        LOOP
          q <= VitalINVIF1( Data, Enable, ResultMap );
          WAIT ON Data, Enable;
        END LOOP;

      ELSE
          -- -------------------------------------
          -- Initialize delay schedules
          -- -------------------------------------
          InvPath ( d_Schd, InitialEdge(Data), tpd_data_q );
          BufEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

        LOOP
          -- -------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- -------------------------------------
          InvPath ( d_Schd, GetEdge(Data), tpd_data_q );
          BufEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );

          -- ------------------------------------
          -- Compute function and propation delay
          -- ------------------------------------
          NewValue := VitalINVIF1( Data, Enable );
          new_schd := NOT d_Schd;

          -- ----------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ----------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                          new_schd, e1_Schd, e0_Schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
```

 Copyright © 2001 IEEE. All rights reserved.

```
                        PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
    END IF;
    END;
--
    PROCEDURE VitalINVIF0 (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
            CONSTANT    ResultMap :  IN VitalResultZMapType
                                        := VitalDefaultResultZMap
    ) IS
        VARIABLE NewValue        : UX01Z;
        VARIABLE new_schd       : SchedType;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE d_Schd, e1_Schd, e0_Schd : SchedType;
        VARIABLE ne1_schd, ne0_schd : SchedType := DefSchedType;
        VARIABLE Dly, Glch      : TIME;
  BEGIN

    -- ------------------------------------------------------------------------
    --  For ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    IF (     (tpd_data_q   = VitalZeroDelay01 )
         AND (tpd_enable_q = VitalZeroDelay01Z)) THEN
      LOOP
        q <= VitalINVIF0( Data, Enable, ResultMap );
        WAIT ON Data, Enable;
      END LOOP;

    ELSE
        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        InvPath ( d_Schd, InitialEdge(Data), tpd_data_q );
        InvEnab ( e1_Schd, e0_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
        InvPath ( d_Schd, GetEdge(Data), tpd_data_q );
        InvEnab ( e1_Schd, e0_Schd, GetEdge(Enable), tpd_enable_q );

        -- ------------------------------------
        -- Compute function and propation delay
        -- ------------------------------------
        NewValue := VitalINVIF0( Data, Enable );
        ne1_schd := NOT e1_Schd;
        ne0_schd := NOT e0_Schd;
        new_schd := NOT d_Schd;

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data),
                        new_schd, ne1_schd, ne0_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                        PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
    END IF;
    END;

    -- ------------------------------------------------------------------------
    -- Multiplexor
    --   MUX   .......... result := data(dselect)
    --   MUX2  .......... 2-input mux; result := data0 when (dselect = '0'),
    --                                       data1 when (dselect = '1'),
    --                        'X' when (dselect = 'X') and (data0 /= data1)
    --   MUX4  .......... 4-input mux; result := data(dselect)
```

```
--   MUX8  .......... 8-input mux; result := data(dselect)
-- ------------------------------------------------------------------------
PROCEDURE VitalMUX2  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL       d1, d0 :  IN std_ulogic;
        SIGNAL         dSel :  IN std_ulogic;
        CONSTANT   tpd_d1_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT   tpd_d0_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_dsel_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
) IS
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
    VARIABLE d1_Schd,  d0_Schd  : SchedType;
    VARIABLE dSel_bSchd, dSel_iSchd : SchedType;
    VARIABLE d1_Edge, d0_Edge, dSel_Edge : EdgeType;
BEGIN

-- ------------------------------------------------------------------------
--  For ALL zero delay paths, use simple model
--   ( No delay selection, glitch detection required )
-- ------------------------------------------------------------------------
IF (     (tpd_d1_q   = VitalZeroDelay01)
     AND (tpd_d0_q   = VitalZeroDelay01)
     AND (tpd_dsel_q = VitalZeroDelay01) ) THEN
  LOOP
    q <= VitalMUX2 ( d1, d0, dSel, ResultMap );
    WAIT ON d1, d0, dSel;
  END LOOP;

ELSE
    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    BufPath ( d1_Schd, InitialEdge(d1), tpd_d1_q );
    BufPath ( d0_Schd, InitialEdge(d0), tpd_d0_q );
    BufPath ( dSel_bSchd, InitialEdge(dSel), tpd_dsel_q );
    InvPath ( dSel_iSchd, InitialEdge(dSel), tpd_dsel_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    BufPath ( d1_Schd, GetEdge(d1), tpd_d1_q );
    BufPath ( d0_Schd, GetEdge(d0), tpd_d0_q );
    BufPath ( dSel_bSchd, GetEdge(dSel), tpd_dsel_q );
    InvPath ( dSel_iSchd, GetEdge(dSel), tpd_dsel_q );

    -- ------------------------------------
    -- Compute function and propation delaq
    -- ------------------------------------
    NewValue  := VitalMUX2 ( d1, d0, dSel );
    new_schd := VitalMUX2 ( d1_Schd, d0_Schd, dSel_bSchd, dSel_iSchd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON d1, d0, dSel;
  END LOOP;
END IF;
END;
--
PROCEDURE VitalMUX4  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL         Data :  IN std_logic_vector4;
        SIGNAL         dSel :  IN std_logic_vector2;
        CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
        CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                 := VitalDefaultResultMap
```

 Copyright © 2001 IEEE. All rights reserved.

```
) IS
    VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
    VARIABLE LastdSel  : std_logic_vector(dSel'RANGE) := (OTHERS=>'U');
    VARIABLE NewValue        : UX01;
    VARIABLE Glitch_Data    : GlitchDataType;
    VARIABLE new_schd       : SchedType;
    VARIABLE Dly, Glch      : TIME;
    VARIABLE Data_Schd      : SchedArray4;
    VARIABLE Data_Edge      : EdgeArray4;
    VARIABLE dSel_Edge      : EdgeArray2;
    VARIABLE dSel_bSchd     : SchedArray2;
    VARIABLE dSel_iSchd     : SchedArray2;
    ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
    ALIAS Atpd_dsel_q : VitalDelayArrayType01(dSel'RANGE) IS tpd_dsel_q;
    VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
BEGIN
  -- ------------------------------------------------------------------------
  --  Check if ALL zero delay paths, use simple model
  --   ( No delay selection, glitch detection required )
  -- ------------------------------------------------------------------------
  FOR i IN dSel'RANGE LOOP
      IF (Atpd_dsel_q(i) /= VitalZeroDelay01) THEN
          AllZeroDelay := FALSE;
          EXIT;
      END IF;
  END LOOP;
  IF (AllZeroDelay) THEN
      FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;

      IF (AllZeroDelay) THEN LOOP
          q <= VitalMUX(Data, dSel, ResultMap);
          WAIT ON Data, dSel;
      END LOOP;
      END IF;
  ELSE

    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    FOR n IN Data'RANGE LOOP
        BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
    END LOOP;
    FOR n IN dSel'RANGE LOOP
        BufPath ( dSel_bSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
        InvPath ( dSel_iSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
    END LOOP;

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    GetEdge ( Data, LastData, Data_Edge );
    BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

    GetEdge ( dSel, LastdSel, dSel_Edge );
    BufPath ( dSel_bSchd, dSel_Edge, Atpd_dsel_q );
    InvPath ( dSel_iSchd, dSel_Edge, Atpd_dsel_q );

    -- ------------------------------------
    -- Compute function and propation delaq
    -- ------------------------------------
    NewValue  := VitalMUX4 ( Data, dSel );
    new_schd := VitalMUX4 ( Data_Schd, dSel_bSchd, dSel_iSchd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON Data, dSel;
```

Copyright © 2001 IEEE. All rights reserved.

```
    END LOOP;
    END IF; --SN
  END;

  PROCEDURE VitalMUX8  (
          SIGNAL            q : OUT std_ulogic;
          SIGNAL         Data :  IN std_logic_vector8;
          SIGNAL         dSel :  IN std_logic_vector3;
          CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
          CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
          CONSTANT  ResultMap :  IN VitalResultMapType  := VitalDefaultResultMap
  ) IS
      VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
      VARIABLE LastdSel  : std_logic_vector(dSel'RANGE) := (OTHERS=>'U');
      VARIABLE NewValue        : UX01;
      VARIABLE Glitch_Data    : GlitchDataType;
      VARIABLE new_schd       : SchedType;
      VARIABLE Dly, Glch      : TIME;
      VARIABLE Data_Schd      : SchedArray8;
      VARIABLE Data_Edge      : EdgeArray8;
      VARIABLE dSel_Edge      : EdgeArray3;
      VARIABLE dSel_bSchd     : SchedArray3;
      VARIABLE dSel_iSchd     : SchedArray3;
      ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
      ALIAS Atpd_dsel_q : VitalDelayArrayType01(dSel'RANGE) IS tpd_dsel_q;
      VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
  BEGIN
    -- ------------------------------------------------------------------------
    --  Check if ALL zero delay paths, use simple model
    --   ( No delay selection, glitch detection required )
    -- ------------------------------------------------------------------------
    FOR i IN dSel'RANGE LOOP
        IF (Atpd_dsel_q(i) /= VitalZeroDelay01) THEN
            AllZeroDelay := FALSE;
            EXIT;
        END IF;
    END LOOP;
    IF (AllZeroDelay) THEN
        FOR i IN Data'RANGE LOOP
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
        END LOOP;

        IF (AllZeroDelay) THEN LOOP
            q <= VitalMUX(Data, dSel, ResultMap);
            WAIT ON Data, dSel;
        END LOOP;
        END IF;
     ELSE

      -- --------------------------------------
      -- Initialize delay schedules
      -- --------------------------------------
      FOR n IN Data'RANGE LOOP
          BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
      END LOOP;
      FOR n IN dSel'RANGE LOOP
          BufPath ( dSel_bSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
          InvPath ( dSel_iSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
      END LOOP;

    LOOP
      -- --------------------------------------
      -- Process input signals
      --   get edge values
      --   re-evaluate output schedules
      -- --------------------------------------
      GetEdge ( Data, LastData, Data_Edge );
      BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

      GetEdge ( dSel, LastdSel, dSel_Edge );
      BufPath ( dSel_bSchd, dSel_Edge, Atpd_dsel_q );
      InvPath ( dSel_iSchd, dSel_Edge, Atpd_dsel_q );

      -- ------------------------------------
      -- Compute function and propation delaq
      -- ------------------------------------
      NewValue  := VitalMUX8 ( Data, dSel );
      new_schd := VitalMUX8 ( Data_Schd, dSel_bSchd, dSel_iSchd );
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, dSel;
      END LOOP;
      END IF;
    END;
--
    PROCEDURE VitalMUX   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            SIGNAL         dSel :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE LastdSel  : std_logic_vector(dSel'RANGE) := (OTHERS=>'U');
        VARIABLE NewValue        : UX01;
        VARIABLE Glitch_Data    : GlitchDataType;
        VARIABLE new_schd       : SchedType;
        VARIABLE Dly, Glch      : TIME;
        VARIABLE Data_Schd      : SchedArray(Data'RANGE);
        VARIABLE Data_Edge      : EdgeArray(Data'RANGE);
        VARIABLE dSel_Edge      : EdgeArray(dSel'RANGE);
        VARIABLE dSel_bSchd     : SchedArray(dSel'RANGE);
        VARIABLE dSel_iSchd     : SchedArray(dSel'RANGE);
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        ALIAS Atpd_dsel_q : VitalDelayArrayType01(dSel'RANGE) IS tpd_dsel_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --    ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      FOR i IN dSel'RANGE LOOP
          IF (Atpd_dsel_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
      END LOOP;
      IF (AllZeroDelay) THEN
          FOR i IN Data'RANGE LOOP
              IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                  AllZeroDelay := FALSE;
                  EXIT;
              END IF;
          END LOOP;

          IF (AllZeroDelay) THEN LOOP
              q <= VitalMUX(Data, dSel, ResultMap);
              WAIT ON Data, dSel;
          END LOOP;
          END IF;
      ELSE

        -- --------------------------------------
        -- Initialize delay schedules
        -- --------------------------------------
        FOR n IN Data'RANGE LOOP
            BufPath ( Data_Schd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;
        FOR n IN dSel'RANGE LOOP
            BufPath ( dSel_bSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
            InvPath ( dSel_iSchd(n), InitialEdge(dSel(n)), Atpd_dsel_q(n) );
        END LOOP;

      LOOP
        -- --------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- --------------------------------------
```

Copyright © 2001 IEEE. All rights reserved.

```
    GetEdge ( Data, LastData, Data_Edge );
    BufPath ( Data_Schd, Data_Edge, Atpd_data_q );

    GetEdge ( dSel, LastdSel, dSel_Edge );
    BufPath ( dSel_bSchd, dSel_Edge, Atpd_dsel_q );
    InvPath ( dSel_iSchd, dSel_Edge, Atpd_dsel_q );

    -- ------------------------------------
    -- Compute function and propation delaq
    -- ------------------------------------
    NewValue  := VitalMUX ( Data, dSel );
    new_schd := VitalMUX ( Data_Schd, dSel_bSchd, dSel_iSchd );

    -- ------------------------------------------------------
    -- Assign Outputs
    --  get delays to new value and possable glitch
    --  schedule output change with On Event glitch detection
    -- ------------------------------------------------------
    GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
    VitalGlitchOnEvent ( q, "q", Glitch_Data, ResultMap(NewValue), Dly,
                         PrimGlitchMode, GlitchDelay=>Glch );

    WAIT ON Data, dSel;
  END LOOP;
 END IF; --SN
END;

-- ---------------------------------------------------------------------------
-- Decoder
--          General Algorithm :
--              (a) Result(...) := '0' when (enable = '0')
--              (b) Result(data) := '1'; all other subelements = '0'
--              ... Result array is decending (n-1 downto 0)
--
--          DECODERn  .......... n:2**n decoder
-- Caution: If 'ResultMap' defines other than strength mapping, the
--          delay selection is not defined.
-- ---------------------------------------------------------------------------
PROCEDURE VitalDECODER2  (
        SIGNAL              q : OUT std_logic_vector2;
        SIGNAL           Data :  IN std_ulogic;
        SIGNAL         Enable :  IN std_ulogic;
        CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
) IS
    VARIABLE NewValue        : std_logic_vector2;
    VARIABLE Glitch_Data    : GlitchArray2;
    VARIABLE new_schd       : SchedArray2;
    VARIABLE Dly, Glch      : TimeArray2;
    VARIABLE Enable_Schd  : SchedType := DefSchedType;
    VARIABLE Data_BSchd, Data_ISchd : SchedType;
BEGIN
  -- -----------------------------------------------------------------------
  --  Check if ALL zero delay paths, use simple model
  --    ( No delay selection, glitch detection required )
  -- -----------------------------------------------------------------------
  IF (tpd_enable_q = VitalZeroDelay01) AND (tpd_data_q = VitalZeroDelay01) THEN
  LOOP
      q <= VitalDECODER2(Data, Enable, ResultMap);
      WAIT ON Data, Enable;
  END LOOP;
  ELSE

    -- --------------------------------------
    -- Initialize delay schedules
    -- --------------------------------------
    BufPath ( Data_BSchd, InitialEdge(Data), tpd_data_q );
    InvPath ( Data_ISchd, InitialEdge(Data), tpd_data_q );
    BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

  LOOP
    -- --------------------------------------
    -- Process input signals
    --   get edge values
    --   re-evaluate output schedules
    -- --------------------------------------
    BufPath ( Data_BSchd, GetEdge(Data), tpd_data_q );
    InvPath ( Data_ISchd, GetEdge(Data), tpd_data_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
          BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

          -- ------------------------------------
          -- Compute function and propation delaq
          -- ------------------------------------
          NewValue  := VitalDECODER2 ( Data, Enable, ResultMap );
          new_schd := VitalDECODER2 ( Data_BSchd, Data_ISchd, Enable_Schd );

          -- ------------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ------------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                               PrimGlitchMode, GlitchDelay=>Glch );

          WAIT ON Data, Enable;
        END LOOP;
        END IF; -- SN
      END;
  --
      PROCEDURE VitalDECODER4  (
              SIGNAL              q : OUT std_logic_vector4;
              SIGNAL           Data :  IN std_logic_vector2;
              SIGNAL         Enable :  IN std_ulogic;
              CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
              CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
             CONSTANT    ResultMap :  IN VitalResultMapType      := VitalDefaultResultMap
      ) IS
          VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
          VARIABLE NewValue        : std_logic_vector4;
          VARIABLE Glitch_Data    : GlitchArray4;
          VARIABLE new_schd       : SchedArray4;
          VARIABLE Dly, Glch      : TimeArray4;
          VARIABLE Enable_Schd    : SchedType;
          VARIABLE Enable_Edge    : EdgeType;
          VARIABLE Data_Edge      : EdgeArray2;
          VARIABLE Data_BSchd, Data_ISchd : SchedArray2;
          ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
          VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
      BEGIN
        -- ------------------------------------------------------------------------
        --  Check if ALL zero delay paths, use simple model
        --   ( No delay selection, glitch detection required )
        -- ------------------------------------------------------------------------
        IF (tpd_enable_q /= VitalZeroDelay01) THEN
            AllZeroDelay := FALSE;
        ELSE
            FOR i IN Data'RANGE LOOP
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
            END LOOP;
        END IF;
        IF (AllZeroDelay) THEN LOOP
            q <= VitalDECODER4(Data, Enable, ResultMap);
            WAIT ON Data, Enable;
        END LOOP;
        ELSE

          -- --------------------------------------
          -- Initialize delay schedules
          -- --------------------------------------
          FOR n IN Data'RANGE LOOP
              BufPath ( Data_BSchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
              InvPath ( Data_ISchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
          END LOOP;
          BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

        LOOP
          -- --------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- --------------------------------------
          GetEdge ( Data, LastData, Data_Edge );
          BufPath ( Data_BSchd, Data_Edge, Atpd_data_q );
          InvPath ( Data_ISchd, Data_Edge, Atpd_data_q );
```

Copyright © 2001 IEEE. All rights reserved.

```
          BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

          -- ------------------------------------
          -- Compute function and propation delaq
          -- ------------------------------------
          NewValue  := VitalDECODER4 ( Data, Enable, ResultMap );
          new_schd := VitalDECODER4 ( Data_BSchd, Data_ISchd, Enable_Schd );

          -- ------------------------------------------------------
          -- Assign Outputs
          --  get delays to new value and possable glitch
          --  schedule output change with On Event glitch detection
          -- ------------------------------------------------------
          GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
          VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                               PrimGlitchMode, GlitchDelay=>Glch );

          WAIT ON Data, Enable;
        END LOOP;
        END IF;
      END;
--
      PROCEDURE VitalDECODER8  (
              SIGNAL              q : OUT std_logic_vector8;
              SIGNAL           Data :  IN std_logic_vector3;
              SIGNAL         Enable :  IN std_ulogic;
              CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
              CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
              CONSTANT    ResultMap :  IN VitalResultMapType
                                       := VitalDefaultResultMap
      ) IS
          VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
          VARIABLE NewValue        : std_logic_vector8;
          VARIABLE Glitch_Data    : GlitchArray8;
          VARIABLE new_schd       : SchedArray8;
          VARIABLE Dly, Glch      : TimeArray8;
          VARIABLE Enable_Schd    : SchedType;
          VARIABLE Enable_Edge    : EdgeType;
          VARIABLE Data_Edge      : EdgeArray3;
          VARIABLE Data_BSchd, Data_ISchd : SchedArray3;
          ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
          VARIABLE AllZeroDelay  : BOOLEAN := TRUE; --SN
      BEGIN
        -- --------------------------------------------------------------------------
        --  Check if ALL zero delay paths, use simple model
        --   ( No delay selection, glitch detection required )
        -- --------------------------------------------------------------------------
        IF (tpd_enable_q /= VitalZeroDelay01) THEN
            AllZeroDelay := FALSE;
        ELSE
            FOR i IN Data'RANGE LOOP
            IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
                AllZeroDelay := FALSE;
                EXIT;
            END IF;
            END LOOP;
        END IF;
        IF (AllZeroDelay) THEN LOOP
            q <= VitalDECODER(Data, Enable, ResultMap);
            WAIT ON Data, Enable;
        END LOOP;
        ELSE

          -- ------------------------------------
          -- Initialize delay schedules
          -- ------------------------------------
          FOR n IN Data'RANGE LOOP
              BufPath ( Data_BSchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
              InvPath ( Data_ISchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
          END LOOP;
          BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

        LOOP
          -- ------------------------------------
          -- Process input signals
          --   get edge values
          --   re-evaluate output schedules
          -- ------------------------------------
          GetEdge ( Data, LastData, Data_Edge );
          BufPath ( Data_BSchd, Data_Edge, Atpd_data_q );
          InvPath ( Data_ISchd, Data_Edge, Atpd_data_q );
```

 Copyright © 2001 IEEE. All rights reserved.

```
        BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

        -- ------------------------------------
        -- Compute function and propation delaq
        -- ------------------------------------
        NewValue  := VitalDECODER8 ( Data, Enable, ResultMap );
        new_schd := VitalDECODER8 ( Data_BSchd, Data_ISchd, Enable_Schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
      END IF; --SN
    END;
--
    PROCEDURE VitalDECODER   (
            SIGNAL              q : OUT std_logic_vector;
            SIGNAL           Data :  IN std_logic_vector;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                     := VitalDefaultResultMap
    ) IS
        VARIABLE LastData  : std_logic_vector(Data'RANGE) := (OTHERS=>'U');
        VARIABLE NewValue        : std_logic_vector(q'RANGE);
        VARIABLE Glitch_Data    : GlitchDataArrayType(q'RANGE);
        VARIABLE new_schd       : SchedArray(q'RANGE);
        VARIABLE Dly, Glch      : VitalTimeArray(q'RANGE);
        VARIABLE Enable_Schd    : SchedType;
        VARIABLE Enable_Edge    : EdgeType;
        VARIABLE Data_Edge      : EdgeArray(Data'RANGE);
        VARIABLE Data_BSchd, Data_ISchd : SchedArray(Data'RANGE);
        ALIAS Atpd_data_q : VitalDelayArrayType01(Data'RANGE) IS tpd_data_q;
        VARIABLE AllZeroDelay  : BOOLEAN := TRUE;
    BEGIN
      -- ------------------------------------------------------------------------
      --  Check if ALL zero delay paths, use simple model
      --    ( No delay selection, glitch detection required )
      -- ------------------------------------------------------------------------
      IF (tpd_enable_q /= VitalZeroDelay01) THEN
          AllZeroDelay := FALSE;
      ELSE
          FOR i IN Data'RANGE LOOP
          IF (Atpd_data_q(i) /= VitalZeroDelay01) THEN
              AllZeroDelay := FALSE;
              EXIT;
          END IF;
          END LOOP;
      END IF;
      IF (AllZeroDelay) THEN LOOP
          q <= VitalDECODER(Data, Enable, ResultMap);
          WAIT ON Data, Enable;
      END LOOP;
      ELSE
        -- ------------------------------------
        -- Initialize delay schedules
        -- ------------------------------------
        FOR n IN Data'RANGE LOOP
            BufPath ( Data_BSchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
            InvPath ( Data_ISchd(n), InitialEdge(Data(n)), Atpd_data_q(n) );
        END LOOP;
        BufPath ( Enable_Schd, InitialEdge(Enable), tpd_enable_q );

      LOOP
        -- ------------------------------------
        -- Process input signals
        --   get edge values
        --   re-evaluate output schedules
        -- ------------------------------------
        GetEdge ( Data, LastData, Data_Edge );
        BufPath ( Data_BSchd, Data_Edge, Atpd_data_q );
        InvPath ( Data_ISchd, Data_Edge, Atpd_data_q );
```


Copyright © 2001 IEEE. All rights reserved.

```
        BufPath ( Enable_Schd, GetEdge(Enable), tpd_enable_q );

        -- -----------------------------------
        -- Compute function and propation delaq
        -- -----------------------------------
        NewValue  := VitalDECODER ( Data, Enable, ResultMap );
        new_schd := VitalDECODER ( Data_BSchd, Data_ISchd, Enable_Schd );

        -- ------------------------------------------------------
        -- Assign Outputs
        --  get delays to new value and possable glitch
        --  schedule output change with On Event glitch detection
        -- ------------------------------------------------------
        GetSchedDelay ( Dly, Glch, NewValue, CurValue(Glitch_Data), new_schd );
        VitalGlitchOnEvent ( q, "q", Glitch_Data, NewValue, Dly,
                             PrimGlitchMode, GlitchDelay=>Glch );

        WAIT ON Data, Enable;
      END LOOP;
      END IF;
    END;

    -- ---------------------------------------------------------------------------
    FUNCTION VitalTruthTable  (
            CONSTANT TruthTable   : IN VitalTruthTableType;
            CONSTANT DataIn       : IN std_logic_vector
          ) RETURN std_logic_vector IS

        CONSTANT InputSize   : INTEGER := DataIn'LENGTH;
        CONSTANT OutSize     : INTEGER := TruthTable'LENGTH(2) - InputSize;
        VARIABLE ReturnValue : std_logic_vector(OutSize - 1 DOWNTO 0)
                             := (OTHERS => 'X');
        VARIABLE DataInAlias : std_logic_vector(0 TO InputSize - 1)
                             :=  To_X01(DataIn);
        VARIABLE Index       : INTEGER;
        VARIABLE Err         : BOOLEAN := FALSE;

        -- This needs to be done since the TableLookup arrays must be
        -- ascending starting with 0
        VARIABLE TableAlias  : VitalTruthTableType(0 TO (TruthTable'LENGTH(1)-1),
                                                   0 TO (TruthTable'LENGTH(2)-1))
                               := TruthTable;

    BEGIN
        -- search through each row of the truth table
        IF OutSize > 0 THEN
          ColLoop:
            FOR i IN TableAlias'RANGE(1) LOOP

            RowLoop: -- Check each input element of the entry
              FOR j IN 0 TO InputSize LOOP

                IF (j = InputSize) THEN -- This entry matches
                    -- Return the Result
                    Index := 0;
                    FOR k IN TruthTable'LENGTH(2) - 1 DOWNTO InputSize LOOP
                        TruthOutputX01Z ( TableAlias(i,k),
                                          ReturnValue(Index), Err);
                        EXIT WHEN Err;
                        Index := Index + 1;
                    END LOOP;

                    IF Err THEN
                        ReturnValue := (OTHERS => 'X');
                    END IF;
                    RETURN ReturnValue;
                END IF;
                IF NOT ValidTruthTableInput(TableAlias(i,j)) THEN
                    VitalError ( "VitalTruthTable", ErrInpSym,
                                 To_TruthChar(TableAlias(i,j)) );
                    EXIT ColLoop;
                END IF;
                EXIT RowLoop WHEN NOT ( TruthTableMatch( DataInAlias(j),
                                                         TableAlias(i, j)));
              END LOOP RowLoop;
            END LOOP ColLoop;

        ELSE
            VitalError ( "VitalTruthTable", ErrTabWidSml );
        END IF;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    RETURN ReturnValue;
END VitalTruthTable;

FUNCTION VitalTruthTable  (
        CONSTANT TruthTable   : IN VitalTruthTableType;
        CONSTANT DataIn       : IN std_logic_vector
      ) RETURN std_logic IS

    CONSTANT InputSize  : INTEGER := DataIn'LENGTH;
    CONSTANT OutSize    : INTEGER := TruthTable'LENGTH(2) - InputSize;
    VARIABLE TempResult : std_logic_vector(OutSize - 1 DOWNTO 0)
                        := (OTHERS => 'X');
BEGIN
    IF (OutSize > 0) THEN
        TempResult := VitalTruthTable(TruthTable, DataIn);
        IF ( 1 > OutSize) THEN
            VitalError ( "VitalTruthTable", ErrTabResSml );
        ELSIF ( 1 < OutSize) THEN
            VitalError ( "VitalTruthTable", ErrTabResLrg );
        END IF;
        RETURN (TempResult(0));
    ELSE
        VitalError ( "VitalTruthTable", ErrTabWidSml );
        RETURN 'X';
    END IF;
END VitalTruthTable;

PROCEDURE VitalTruthTable (
        SIGNAL   Result     : OUT std_logic_vector;
        CONSTANT TruthTable : IN  VitalTruthTableType;
        SIGNAL   DataIn     : IN  std_logic_vector        -- IR#236
) IS
    CONSTANT ResLeng     : INTEGER := Result'LENGTH;
    CONSTANT ActResLen   : INTEGER := TruthTable'LENGTH(2) - DataIn'LENGTH;
    CONSTANT FinalResLen : INTEGER := Minimum(ActResLen, ResLeng);
    VARIABLE TempResult  : std_logic_vector(ActResLen - 1 DOWNTO 0)
                         := (OTHERS => 'X');

BEGIN
    TempResult := VitalTruthTable(TruthTable, DataIn);

    IF (ResLeng > ActResLen) THEN
        VitalError ( "VitalTruthTable", ErrTabResSml );
    ELSIF (ResLeng < ActResLen) THEN
        VitalError ( "VitalTruthTable", ErrTabResLrg );
    END IF;
    TempResult(FinalResLen-1 DOWNTO 0) := TempResult(FinalResLen-1 DOWNTO 0);
    Result <= TempResult;

END VitalTruthTable;

PROCEDURE VitalTruthTable (
        SIGNAL   Result     : OUT std_logic;
        CONSTANT TruthTable : IN VitalTruthTableType;
        SIGNAL DataIn       : IN std_logic_vector        -- IR#236
) IS

    CONSTANT ActResLen  : INTEGER := TruthTable'LENGTH(2) - DataIn'LENGTH;
    VARIABLE TempResult : std_logic_vector(ActResLen - 1 DOWNTO 0)
                        := (OTHERS => 'X');

BEGIN
    TempResult := VitalTruthTable(TruthTable, DataIn);

    IF ( 1 > ActResLen) THEN
        VitalError ( "VitalTruthTable", ErrTabResSml );
    ELSIF ( 1 < ActResLen) THEN
        VitalError ( "VitalTruthTable", ErrTabResLrg );
    END IF;
    IF (ActResLen >  0) THEN
        Result <= TempResult(0);
    END IF;

END VitalTruthTable;

-- ------------------------------------------------------------------------
PROCEDURE VitalStateTable (
        VARIABLE Result         : INOUT std_logic_vector;
        VARIABLE PreviousDataIn : INOUT std_logic_vector;
        CONSTANT StateTable     : IN VitalStateTableType;
        CONSTANT DataIn         : IN std_logic_vector;
```


Copyright © 2001 IEEE. All rights reserved.

```
        CONSTANT NumStates      : IN NATURAL
) IS

    CONSTANT InputSize     : INTEGER := DataIn'LENGTH;
    CONSTANT OutSize       : INTEGER
                             := StateTable'LENGTH(2) - InputSize - NumStates;
    CONSTANT ResLeng       : INTEGER := Result'LENGTH;
    VARIABLE DataInAlias   : std_logic_vector(0 TO DataIn'LENGTH-1)
                             := To_X01(DataIn);
    VARIABLE PrevDataAlias : std_logic_vector(0 TO PreviousDataIn'LENGTH-1)
                             := To_X01(PreviousDataIn);
    VARIABLE ResultAlias   : std_logic_vector(0 TO ResLeng-1)
                             := To_X01(Result);
    VARIABLE ExpResult     : std_logic_vector(0 TO OutSize-1);

BEGIN
    IF (PreviousDataIn'LENGTH < DataIn'LENGTH) THEN
        VitalError ( "VitalStateTable", ErrVctLng, "PreviousDataIn<DataIn");

        ResultAlias := (OTHERS => 'X');
        Result := ResultAlias;

    ELSIF (OutSize <= 0) THEN
        VitalError ( "VitalStateTable", ErrTabWidSml );

        ResultAlias := (OTHERS => 'X');
        Result := ResultAlias;

    ELSE
        IF (ResLeng > OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResSml );
        ELSIF (ResLeng < OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResLrg );
        END IF;

        ExpResult := StateTableLookUp ( StateTable, DataInAlias,
                                        PrevDataAlias, NumStates,
                                        ResultAlias);
        ResultAlias := (OTHERS => 'X');
        ResultAlias ( Maximum(0, ResLeng - OutSize) TO ResLeng - 1)
               := ExpResult(Maximum(0, OutSize - ResLeng) TO OutSize-1);

        Result := ResultAlias;
        PrevDataAlias(0 TO InputSize - 1) := DataInAlias;
        PreviousDataIn := PrevDataAlias;

    END IF;
END VitalStateTable;


PROCEDURE VitalStateTable (
        VARIABLE Result         : INOUT std_logic;        -- states
     VARIABLE PreviousDataIn : INOUT std_logic_vector; -- previous inputs and states
       CONSTANT StateTable     : IN VitalStateTableType; -- User's StateTable data
        CONSTANT DataIn         : IN std_logic_vector     -- Inputs
) IS

    VARIABLE ResultAlias : std_logic_vector(0 TO 0);
BEGIN
    ResultAlias(0) := Result;
    VitalStateTable ( StateTable     => StateTable,
                      DataIn         => DataIn,
                      NumStates      => 1,
                      Result         => ResultAlias,
                      PreviousDataIn => PreviousDataIn
                    );
    Result := ResultAlias(0);

END VitalStateTable;

PROCEDURE VitalStateTable (
        SIGNAL   Result     : INOUT std_logic_vector;
        CONSTANT StateTable : IN VitalStateTableType;
        SIGNAL   DataIn     : IN std_logic_vector;
        CONSTANT NumStates  : IN NATURAL
) IS

    CONSTANT InputSize   : INTEGER := DataIn'LENGTH;
    CONSTANT OutSize     : INTEGER
                           := StateTable'LENGTH(2) - InputSize - NumStates;
    CONSTANT ResLeng     : INTEGER := Result'LENGTH;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE PrevData    : std_logic_vector(0 TO DataIn'LENGTH-1)
                         := (OTHERS => 'X');
    VARIABLE DataInAlias : std_logic_vector(0 TO DataIn'LENGTH-1);
    VARIABLE ResultAlias : std_logic_vector(0 TO ResLeng-1);
    VARIABLE ExpResult   : std_logic_vector(0 TO OutSize-1);

BEGIN
    IF (OutSize <= 0) THEN
        VitalError ( "VitalStateTable", ErrTabWidSml );

        ResultAlias  := (OTHERS => 'X');
        Result <= ResultAlias;

    ELSE
        IF (ResLeng > OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResSml );
        ELSIF (ResLeng < OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResLrg );
        END IF;

        LOOP
            DataInAlias := To_X01(DataIn);
            ResultAlias := To_X01(Result);
            ExpResult   := StateTableLookUp ( StateTable, DataInAlias,
                                              PrevData, NumStates,
                                              ResultAlias);
            ResultAlias := (OTHERS => 'X');
            ResultAlias(Maximum(0, ResLeng - OutSize) TO ResLeng-1)
                   := ExpResult(Maximum(0, OutSize - ResLeng) TO OutSize-1);

            Result   <= ResultAlias;
            PrevData := DataInAlias;

            WAIT ON DataIn;
        END LOOP;

    END IF;

END VitalStateTable;

PROCEDURE VitalStateTable (
        SIGNAL   Result     : INOUT std_logic;
        CONSTANT StateTable : IN VitalStateTableType;
        SIGNAL   DataIn     : IN std_logic_vector
) IS

    CONSTANT InputSize   : INTEGER := DataIn'LENGTH;
    CONSTANT OutSize     : INTEGER := StateTable'LENGTH(2) - InputSize-1;

    VARIABLE PrevData    : std_logic_vector(0 TO DataIn'LENGTH-1)
                         := (OTHERS => 'X');
    VARIABLE DataInAlias : std_logic_vector(0 TO DataIn'LENGTH-1);
    VARIABLE ResultAlias : std_logic_vector(0 TO 0);
    VARIABLE ExpResult   : std_logic_vector(0 TO OutSize-1);

BEGIN
    IF (OutSize <= 0) THEN
        VitalError ( "VitalStateTable", ErrTabWidSml );

        Result <= 'X';

    ELSE
        IF ( 1 > OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResSml );
        ELSIF ( 1 < OutSize) THEN
            VitalError ( "VitalStateTable", ErrTabResLrg );
        END IF;

        LOOP
            ResultAlias(0) := To_X01(Result);
            DataInAlias := To_X01(DataIn);
            ExpResult   := StateTableLookUp ( StateTable, DataInAlias,
                                              PrevData, 1, ResultAlias);

            Result   <= ExpResult(OutSize-1);
            PrevData := DataInAlias;

            WAIT ON DataIn;
        END LOOP;
    END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
    END VitalStateTable;

    -- -------------------------------------------------------------------------
    -- std_logic resolution primitive
    -- -------------------------------------------------------------------------
    PROCEDURE VitalResolve (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector  --IR236 4/2/98
    ) IS
        VARIABLE uData : std_ulogic_vector(Data'RANGE);
    BEGIN
        FOR i IN Data'RANGE LOOP
            uData(i) := Data(i);
        END LOOP;
        q <= resolved(uData);
    END;

END VITAL_Primitives;
```

13.5 VITAL_Memory package declaration

```
-- ----------------------------------------------------------------------------
-- Title        : Standard VITAL Memory Package
--              :
-- Library      : Vital_Memory
--              :
-- Developers   : IEEE DASC Timing Working Group (TWG), PAR 1076.4
--              : Ekambaram Balaji, LSI Logic Corporation
--              : Jose De Castro, Consultant
--              : Prakash Bare, GDA Technologies
--              : William Yam, LSI Logic Corporation
--              : Dennis Brophy, Model Technology
--              :
-- Purpose      : This packages defines standard types, constants, functions
--              : and procedures for use in developing ASIC memory models.
--              :
-- ----------------------------------------------------------------------------
--
-- ----------------------------------------------------------------------------
-- Modification History :
-- ----------------------------------------------------------------------------
-- Ver:|Auth:| Date:| Changes Made:
-- 0.1 | eb  |071796| First prototye as part of VITAL memory proposal
-- 0.2 | jdc |012897| Initial prototyping with proposed MTM scheme
-- 0.3 | jdc |090297| Extensive updates for TAG review (functional)
-- 0.4 | eb  |091597| Changed naming conventions for VitalMemoryTable
--     |     |      | Added interface of VitalMemoryCrossPorts() &
--     |     |      | VitalMemoryViolation().
-- 0.5 | jdc |092997| Completed naming changes thoughout package body.
--     |     |      | Testing with simgle port test model looks ok.
-- 0.6 | jdc |121797| Major updates to the packages:
--     |     |      | - Implement VitalMemoryCrossPorts()
--     |     |      |   - Use new VitalAddressValueType
--     |     |      |   - Use new VitalCrossPortModeType enum
--     |     |      |   - Overloading without SamePort args
--     |     |      |   - Honor erroneous address values
--     |     |      |   - Honor ports disabled with 'Z'
--     |     |      | - Implement implicit read 'M' table symbol
--     |     |      | - Cleanup buses to use (H DOWNTO L)
--     |     |      | - Message control via MsgOn,HeaderMsg,PortName
--     |     |      | - Tested with 1P1RW,2P2RW,4P2R2W,4P4RW cases
-- 0.7 | jdc |052698| Bug fixes to the packages:
--     |     |      | - Fix failure with negative Address values
--     |     |      | - Added debug messages for VMT table search
--     |     |      | - Remove 'S' for action column (only 's')
--     |     |      | - Remove 's' for response column (only 'S')
--     |     |      | - Remove 'X' for action and response columns
-- 0.8 | jdc |061298| Implemented VitalMemoryViolation()
--     |     |      | - Minimal functionality violation tables
--     |     |      | - Missing:
--     |     |      |   - Cannot handle wide violation variables
--     |     |      |   - Cannot handle sub-word cases
--     |     |      | Fixed IIC version of MemoryMatch
--     |     |      | Fixed 'M' vs 'm' switched on debug output
--     |     |      | TO BE DONE:
--     |     |      | - Implement 'd' corrupting a single bit
```

 Copyright © 2001 IEEE. All rights reserved.

```
--      |     |      | - Implement 'D' corrupting a single bit
-- 0.9 |eb/sc|080498| Added UNDEF value for VitalPortFlagType
-- 0.10|eb/sc|080798| Added CORRUPT value for VitalPortFlagType
-- 0.11|eb/sc|081798| Added overloaded function interface for
--     |     |      |       VitalDeclareMemory
-- 0.14| jdc |113198| Merging of memory functionality and version
--     |     |      | 1.4 9/17/98 of timing package from Prakash
-- 0.15| jdc |120198| Major development of VMV functionality
-- 0.16| jdc |120298| Complete VMV functionlality for initial testing
--     |     |      | - New ViolationTableCorruptMask() procedure
--     |     |      | - New MemoryTableCorruptMask() procedure
--     |     |      | - HandleMemoryAction():
--     |     |      |   - Removed DataOutBus bogus output
--     |     |      |   - Replaced DataOutTmp with DataInTmp
--     |     |      |   - Added CorruptMask input handling
--     |     |      |   - Implemented 'd','D' using CorruptMask
--     |     |      |   - CorruptMask on 'd','C','L','D','E'
--     |     |      |   - CorruptMask ignored on 'c','l','e'
--     |     |      |   - Changed 'l','d','e' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'L','D','E' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'c','l','d','e' to ignore HighBit, LowBit
--     |     |      |   - Changed 'C','L','D','E' to use HighBit, LowBit
--     |     |      | - HandleDataAction():
--     |     |      |   - Added CorruptMask input handling
--     |     |      |   - Implemented 'd','D' using CorruptMask
--     |     |      |   - CorruptMask on 'd','C','L','D','E'
--     |     |      |   - CorruptMask ignored on 'l','e'
--     |     |      |   - Changed 'l','d','e' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'L','D','E' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'l','d','e' to ignore HighBit, LowBit
--     |     |      |   - Changed 'L','D','E' to use HighBit, LowBit
--     |     |      | - MemoryTableLookUp():
--     |     |      |   - Added MsgOn table debug output
--     |     |      |   - Uses new MemoryTableCorruptMask()
--     |     |      | - ViolationTableLookUp():
--     |     |      |   - Uses new ViolationTableCorruptMask()
-- 0.17| jdc |120898| - Added VitalMemoryViolationSymbolType,
--     |     |      |   VitalMemoryViolationTableType data
--     |     |      |   types but not used yet (need to discuss)
--     |     |      | - Added overload for VitalMemoryViolation()
--     |     |      |   which does not have array flags
--     |     |      | - Bug fixes for VMV functionality:
--     |     |      |   - ViolationTableLookUp() not handling '-' in
--     |     |      |     scalar violation matching
--     |     |      |   - VitalMemoryViolation() now normalizes
--     |     |      |     VFlagArrayTmp'LEFT as LSB before calling
--     |     |      |     ViolationTableLookUp() for proper scanning
--     |     |      |   - ViolationTableCorruptMask() had to remove
--     |     |      |     normalization of CorruptMaskTmp and
--     |     |      |     ViolMaskTmp for proper MSB:LSB corruption
--     |     |      |   - HandleMemoryAction(), HandleDataAction()
--     |     |      |     - Removed 'D','E' since not being used
--     |     |      |     - Use XOR instead of OR for corrupt masks
--     |     |      |     - Now 'd' is sensitive to HighBit, LowBit
--     |     |      |   - Fixed LowBit overflow in bit writeable case
--     |     |      |     - MemoryTableCorruptMask()
--     |     |      |     - ViolationTableCorruptMask()
--     |     |      |     - VitalMemoryTable()
--     |     |      |     - VitalMemoryCrossPorts()
--     |     |      |   - Fixed VitalMemoryViolation() failing on
--     |     |      |     error AddressValue from earlier VMT()
--     |     |      | - Minor cleanup of code formatting
-- 0.18| jdc |032599| - In VitalDeclareMemory()
--     |     |      |   - Added BinaryLoadFile formal arg and
--     |     |      |     modified LoadMemory() to handle bin
--     |     |      | - Added NOCHANGE to VitalPortFlagType
--     |     |      | - For VitalCrossPortModeType
--     |     |      |   - Added CpContention enum
--     |     |      | - In HandleDataAction()
--     |     |      |   - Set PortFlag := NOCHANGE for 'S'
--     |     |      | - In HandleMemoryAction()
--     |     |      |   - Set PortFlag := NOCHANGE for 's'
--     |     |      | - In VitalMemoryTable() and
--     |     |      |   VitalMemoryViolation()
--     |     |      |   - Honor PortFlag = NOCHANGE returned
--     |     |      |     from HandleMemoryAction()
--     |     |      | - In VitalMemoryCrossPorts()
--     |     |      |   - Fixed Address = AddressJ for all
--     |     |      |     conditions of DoWrCont & DoCpRead
--     |     |      |   - Handle CpContention like WrContOnly
--     |     |      |     under CpReadOnly conditions, with
```

Copyright © 2001 IEEE. All rights reserved.

```
--      |      |       |     associated memory message changes
--      |      |       |   - Handle PortFlag = NOCHANGE like
--      |      |       |     PortFlag = READ for actions
--      |      |       | - Modeling change:
--      |      |       |   - Need to init PortFlag every delta
--      |      |       |     PortFlag_A := (OTHES => UNDEF);
--      |      |       | - Updated InternalTimingCheck code
-- 0.19| jdc  |042599| - Fixes for bit-writeable cases
--      |      |       | - Check PortFlag after HandleDataAction
--      |      |       |   in VitalMemoryViolation()
-- 0.20| jdc  |042599| - Merge PortFlag changes from Prakash
--      |      |       |   and Willian:
--      |      |       |     VitalMemorySchedulePathDelay()
--      |      |       |     VitalMemoryExpandPortFlag()
-- 0.21| jdc  |072199| - Changed VitalCrossPortModeType enums,
--      |      |       |   added new CpReadAndReadContention.
--      |      |       | - Fixed VitalMemoryCrossPorts() parameter
--      |      |       |   SamePortFlag to INOUT so that it can
--      |      |       |   set CORRUPT or READ value.
--      |      |       | - Fixed VitalMemoryTable() where PortFlag
--      |      |       |   setting by HandleDataAction() is being
--      |      |       |   ignored when HandleMemoryAction() sets
--      |      |       |   PortFlagTmp to NOCHANGE.
--      |      |       | - Fixed VitalMemoryViolation() to set
--      |      |       |   all bits of PortFlag when violating.
-- 0.22| jdc  |072399| - Added HIGHZ to PortFlagType. HandleData
--      |      |       |   checks whether the previous state is HIGHZ.
--      |      |       |   If yes then portFlag should be NOCHANGE
--      |      |       |   for VMPD to ignore IORetain corruption.
--      |      |       |   The idea is that the first Z should be
--      |      |       |   propagated but later ones should be ignored.
--      |      |       |
-- 0.23| jdc  |100499| - Took code checked in by Dennis 09/28/99
--      |      |       | - Changed VitalPortFlagType to record of
--      |      |       |   new VitalPortStateType to hold current,
--      |      |       |   previous values and separate disable.
--      |      |       |   Also created VitalDefaultPortFlag const.
--      |      |       |   Removed usage of PortFlag NOCHANGE
--      |      |       | - VitalMemoryTable() changes:
--      |      |       |   Optimized return when all curr = prev
--      |      |       |   AddressValue is now INOUT to optimize
--      |      |       |   Transfer PF.MemoryCurrent to MemoryPrevious
--      |      |       |   Transfer PF.DataCurrent to DataPrevious
--      |      |       |   Reset PF.OutputDisable to FALSE
--      |      |       |   Expects PortFlag init in declaration
--      |      |       |   No need to init PortFlag every delta
--      |      |       | - VitalMemorySchedulePathDelay() changes:
--      |      |       |   Initialize with VitalDefaultPortFlag
--      |      |       |   Check PortFlag.OutputDisable
--      |      |       | - HandleMemoryAction() changes:
--      |      |       |   Set value of PortFlag.MemoryCurrent
--      |      |       |   Never set PortFlag.OutputDisable
--      |      |       | - HandleDataAction() changes:
--      |      |       |   Set value of PortFlag.DataCurrent
--      |      |       |   Set PortFlag.DataCurrent for HIGHZ
--      |      |       | - VitalMemoryCrossPorts() changes:
--      |      |       |   Check/set value of PF.MemoryCurrent
--      |      |       |   Check value of PF.OutputDisable
--      |      |       | - VitalMemoryViolation() changes:
--      |      |       |   Fixed bug - not reading inout PF value
--      |      |       |   Clean up setting of PortFlag
-- 0.24| jdc  |100899| - Modified update of PF.OutputDisable
--      |      |       |   to correctly accomodate 2P1W1R case:
--      |      |       |   the read port should not exhibit
--      |      |       |   IO retain corrupt when reading
--      |      |       |   addr unrelated to addr being written.
-- 0.25| jdc  |100999| - VitalMemoryViolation() change:
--      |      |       |   Fixed bug with RDNWR mode incorrectly
--      |      |       |   updating the PF.OutputDisable
-- 0.26| jdc  |100999| - VitalMemoryCrossPorts() change:
--      |      |       |   Fixed bugs with update of PF
-- 0.27| jdc  |101499| - VitalMemoryCrossPorts() change:
--      |      |       |   Added DoRdWrCont message (ErrMcpRdWrCo,
--      |      |       |   Memory cross port read/write data only
--      |      |       |   contention)
--      |      |       | - VitalMemoryTable() change:
--      |      |       |   Set PF.OutputDisable := TRUE for the
--      |      |       |   optimized cases.
-- 0.28| pb   |112399| - Added 8 VMPD procedures for vector
--      |      |       |   PathCondition support. Now the total
--      |      |       |   number of overloadings for VMPD is 24.
```

 Copyright © 2001 IEEE. All rights reserved.

```
--     |     |      | - Number of overloadings for SetupHold
--     |     |      |   procedures increased to 5. Scalar violations
--     |     |      |   are not supported anymore. Vector checkEnabled
--     |     |      |   support is provided through the new overloading
-- 0.29| jdc |120999| - HandleMemoryAction() HandleDataAction()
--     |     |      |   Reinstated 'D' and 'E' actions but
--     |     |      |   with new PortFlagType
--     |     |      | - Updated file handling syntax, must compile
--     |     |      |   with -93 syntax now.
-- 0.30| jdc |022300| - Formated for 80 column max width
-- ----------------------------------------------------------------------------

LIBRARY IEEE;
USE     IEEE.STD_LOGIC_1164.ALL;
USE     IEEE.Vital_Timing.ALL;
USE     IEEE.Vital_Primitives.ALL;

LIBRARY STD;
USE     STD.TEXTIO.ALL;

PACKAGE Vital_Memory IS

-- ----------------------------------------------------------------------------
-- Timing Section
-- ----------------------------------------------------------------------------

-- ----------------------------------------------------------------------------
-- Types and constants for Memory timing procedures
-- ----------------------------------------------------------------------------
TYPE VitalMemoryArcType       IS (ParallelArc, CrossArc, SubwordArc);
TYPE OutputRetainBehaviorType IS (BitCorrupt, WordCorrupt);
TYPE VitalMemoryMsgFormatType IS (Vector, Scalar, VectorEnum);
TYPE X01ArrayT IS ARRAY (NATURAL RANGE <> ) OF X01;
TYPE X01ArrayPT IS ACCESS X01ArrayT;
TYPE VitalMemoryViolationType IS ACCESS X01ArrayT;
CONSTANT DefaultNumBitsPerSubword : INTEGER := -1;


-- Data type storing path delay and schedule information for output bits
TYPE VitalMemoryScheduleDataType IS RECORD
  OutputData        : std_ulogic;
  NumBitsPerSubWord : INTEGER;
  ScheduleTime      : TIME;
  ScheduleValue     : std_ulogic;
  LastOutputValue   : std_ulogic;
  PropDelay         : TIME;
  OutputRetainDelay : TIME;
  InputAge          : TIME;
END RECORD;

TYPE VitalMemoryTimingDataType IS RECORD
  NotFirstFlag : BOOLEAN;
  RefLast      : X01;
  RefTime      : TIME;
  HoldEn       : BOOLEAN;
  TestLast     : std_ulogic;
  TestTime     : TIME;
  SetupEn      : BOOLEAN;
  TestLastA    : VitalLogicArrayPT;
  TestTimeA    : VitalTimeArrayPT;
  RefLastA     : X01ArrayPT;
  RefTimeA     : VitalTimeArrayPT;
  HoldEnA      : VitalBoolArrayPT;
  SetupEnA     : VitalBoolArrayPT;
END RECORD;

TYPE VitalPeriodDataArrayType IS ARRAY (NATURAL RANGE <>) OF
  VitalPeriodDataType;

-- Data type storing path delay and schedule information for output
-- vectors
TYPE VitalMemoryScheduleDataVectorType IS ARRAY (NATURAL RANGE <> ) OF
  VitalMemoryScheduleDataType;

-- VitalPortFlagType records runtime mode of port sub-word slices
--    TYPE VitalPortFlagType IS (
--        UNDEF,
--        READ,
--        WRITE,
--        CORRUPT,
--        HIGHZ,
```


Copyright © 2001 IEEE. All rights reserved.

```
    --        NOCHANGE
    --    );

    -- VitalPortFlagType records runtime mode of port sub-word slices
    TYPE VitalPortStateType IS (
      UNDEF,
      READ,
      WRITE,
      CORRUPT,
      HIGHZ
    );

    TYPE VitalPortFlagType IS RECORD
      MemoryCurrent   : VitalPortStateType;
      MemoryPrevious  : VitalPortStateType;
      DataCurrent     : VitalPortStateType;
      DataPrevious    : VitalPortStateType;
      OutputDisable   : BOOLEAN;
    END RECORD;

    CONSTANT VitalDefaultPortFlag : VitalPortFlagType := (
      MemoryCurrent   => READ,
      MemoryPrevious  => UNDEF,
      DataCurrent     => READ,
      DataPrevious    => UNDEF,
      OutputDisable   => FALSE
    );

    -- VitalPortFlagVectorType to be same width i as enables of a port
    -- or j multiples thereof, where j is the number of cross ports
    TYPE VitalPortFlagVectorType IS
      ARRAY (NATURAL RANGE <>) OF VitalPortFlagType;

    -- ------------------------------------------------------------------------------
    -- Functions  : VitalMemory path delay procedures
    --              - VitalMemoryInitPathDelay
    --              - VitalMemoryAddPathDelay
    --              - VitalMemorySchedulePathDelay
    --
    -- Description: VitalMemoryInitPathDelay, VitalMemoryAddPathDelay and
    --             VitalMemorySchedulePathDelay are Level 1 routines used
    --             for selecting the propagation delay paths based on
    --             path condition, transition type and delay values and
    --             schedule a new output value.
    --
    --             Following features are implemented in these procedures:
    --             o condition dependent path selection
    --             o Transition dependent delay selection
    --             o shortest delay path selection from multiple
    --               candidate paths
    --             o Scheduling of the computed values on the specified
    --               signal.
    --             o output retain behavior if outputRetain flag is set
    --             o output mapping to alternate strengths to model
    --               pull-up, pull-down etc.
    --
    --             <More details to be added here>
    --
    -- Following is information on overloading of the procedures.
    --
    -- VitalMemoryInitPathDelay is overloaded for ScheduleDataArray and
    -- OutputDataArray
    --
    -- ------------------------------------------------------------------------------
    --          ScheduleDataArray        OutputDataArray
    -- ------------------------------------------------------------------------------
    --             Scalar                    Scalar
    --             Vector                    Vector
    -- ------------------------------------------------------------------------------
    --
    --
    -- VitalMemoryAddPathDelay is overloaded for ScheduleDataArray,
    -- PathDelayArray, InputSignal and delaytype.
    --
    -- ------------------------------------------------------------------------------
    --   DelayType        InputSignal    ScheduleData     PathDelay
    --                                      Array          Array
    -- ------------------------------------------------------------------------------
    --  VitalDelayType      Scalar         Scalar          Scalar
    --  VitalDelayType      Scalar         Vector          Vector
    --  VitalDelayType      Vector         Scalar          Vector
```

 Copyright © 2001 IEEE. All rights reserved.

```
--  VitalDelayType      Vector         Vector          Vector
--  VitalDelayType01    Scalar         Scalar          Scalar
--  VitalDelayType01    Scalar         Vector          Vector
--  VitalDelayType01    Vector         Scalar          Vector
--  VitalDelayType01    Vector         Vector          Vector
--  VitalDelayType01Z   Scalar         Scalar          Scalar
--  VitalDelayType01Z   Scalar         Vector          Vector
--  VitalDelayType01Z   Vector         Scalar          Vector
--  VitalDelayType01Z   Vector         Vector          Vector
--  VitalDelayType01XZ  Scalar         Scalar          Scalar
--  VitalDelayType01XZ  Scalar         Vector          Vector
--  VitalDelayType01XZ  Vector         Scalar          Vector
--  VitalDelayType01XZ  Vector         Vector          Vector
-- ----------------------------------------------------------------------------
--
--
--  VitalMemorySchedulePathDelay is overloaded for ScheduleDataArray,
--  and OutSignal
--
-- ----------------------------------------------------------------------------
--          OutSignal                ScheduleDataArray
-- ----------------------------------------------------------------------------
--            Scalar                     Scalar
--            Vector                     Vector
-- ----------------------------------------------------------------------------
--
-- Procedure Declarations:
--
--
-- Function     :   VitalMemoryInitPathDelay
--
-- Arguments:
--
--  INOUT               Type             Description
--
--  ScheduleDataArray/  VitalMemoryScheduleDataVectorType/
--  ScheduleData        VitalMemoryScheduleDataType
--                                       Internal data variable for
--                                       storing delay and schedule
--                                       information for each output bit
--
--
--  IN
--
--  OutputDataArray/    STD_LOGIC_VECTOR/Array containing current output
--  OutputData          STD_ULOGIC       value
--
--
--  NumBitsPerSubWord   INTEGER          Number of bits per subword.
--                                       Default value of this argument
--                                       is DefaultNumBitsPerSubword
--                                       which is interpreted as no
--                                       subwords
--
-- ----------------------------------------------------------------------------
--
--
-- ScheduleDataArray - Vector
-- OutputDataArray - Vector
--
PROCEDURE VitalMemoryInitPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  VARIABLE OutputDataArray   : IN STD_LOGIC_VECTOR;
  CONSTANT NumBitsPerSubWord : IN INTEGER := DefaultNumBitsPerSubword
);
--
-- ScheduleDataArray - Scalar
-- OutputDataArray - Scalar
--
PROCEDURE VitalMemoryInitPathDelay (
  VARIABLE ScheduleData      : INOUT VitalMemoryScheduleDataType;
  VARIABLE OutputData        : IN STD_ULOGIC
);

-- ----------------------------------------------------------------------------
--
-- Function     :   VitalMemoryAddPathDelay
--
-- Arguments
--
--  INOUT                 Type            Description
```

Copyright © 2001 IEEE. All rights reserved.

```
--
--  ScheduleDataArray/    VitalMemoryScheduleDataVectorType/
--  ScheduleData          VitalMemoryScheduleDataType
--                                        Internal data variable for
--                                        storing delay and schedule
--                                        information for each output bit
--
--  InputChangeTimeArray/ VitaltimeArrayT/Time
--  InputChangeTime                       Holds the time since the last
--                                        input change
--
--  IN
--
--  InputSignal           STD_LOGIC_VECTOR
--                        STD_ULOGIC/     Array holding the input value
--
--  OutputSignalName      STRING          The output signal name
--
--  PathDelayArray/       VitalDelayArrayType01ZX,
--  PathDelay             VitalDelayArrayType01Z,
--                        VitalDelayArrayType01,
--                        VitalDelayArrayType/
--                        VitalDelayType01ZX,
--                        VitalDelayType01Z,
--                        VitalDelayType01,
--                        VitalDelayType  Array of delay values
--
--  ArcType               VitalMemoryArcType
--                                        Indicates the Path type. This
--                                        can be SubwordArc, CrossArc or
--                                        ParallelArc
--
--  PathCondition         BOOLEAN         If True, the transition in
--                                        the corresponding input signal
--                                        is considered while
--                                        caluculating the prop. delay
--                                        else the transition is ignored.
--
--  OutputRetainFlag      BOOLEAN         If specified TRUE,output retain
--                                        (hold) behavior is implemented.
--
-- ----------------------------------------------------------------------------
--
-- #1
-- DelayType - VitalDelayType
-- Input     - Scalar
-- Output    - Scalar
-- Delay     - Scalar
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_ULOGIC;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTime      : INOUT Time;
  CONSTANT PathDelay            : IN VitalDelayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE
);

-- #2
-- DelayType - VitalDelayType
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE
);

-- #3
-- DelayType - VitalDelayType
-- Input     - Scalar
-- Output    - Vector
```


Copyright © 2001 IEEE. All rights reserved.

```
-- Delay     - Vector
-- Condition - Vector

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT
);

-- #4
-- DelayType - VitalDelayType
-- Input     - Vector
-- Output    - Scalar
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE
);

-- #5
-- DelayType - VitalDelayType
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray    : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE
);

-- #6
-- DelayType - VitalDelayType
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray    : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT
);

-- #7
-- DelayType - VitalDelayType01
-- Input     - Scalar
-- Output    - Scalar
-- Delay     - Scalar
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_ULOGIC;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTime      : INOUT Time;
  CONSTANT PathDelay            : IN VitalDelayType01;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE
);
```

Copyright © 2001 IEEE. All rights reserved.

```
-- #8
-- DelayType - VitalDelayType01
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE
);

-- #9
-- DelayType - VitalDelayType01
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT
);

-- #10
-- DelayType - VitalDelayType01
-- Input     - Vector
-- Output    - Scalar
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE
);

-- #11
-- DelayType - VitalDelayType01
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE
);

-- #12
-- DelayType - VitalDelayType01
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
```

 Copyright © 2001 IEEE. All rights reserved.

```
      CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
      CONSTANT PathConditionArray   : IN VitalBoolArrayT
    );

    -- #13
    -- DelayType - VitalDelayType01Z
    -- Input     - Scalar
    -- Output    - Scalar
    -- Delay     - Scalar
    -- Condition - Scalar

    PROCEDURE VitalMemoryAddPathDelay (
      VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
      SIGNAL   InputSignal          : IN STD_ULOGIC;
      CONSTANT OutputSignalName     : IN STRING := "";
      VARIABLE InputChangeTime      : INOUT Time;
      CONSTANT PathDelay            : IN VitalDelayType01Z;
      CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
      CONSTANT PathCondition        : IN BOOLEAN := TRUE;
      CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE
    );

    -- #14
    -- DelayType - VitalDelayType01Z
    -- Input     - Scalar
    -- Output    - Vector
    -- Delay     - Vector
    -- Condition - Scalar

    PROCEDURE VitalMemoryAddPathDelay (
      VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
      SIGNAL   InputSignal       : IN STD_ULOGIC;
      CONSTANT OutputSignalName  : IN STRING := "";
      VARIABLE InputChangeTime   : INOUT Time;
      CONSTANT PathDelayArray    : IN VitalDelayArrayType01Z;
      CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
      CONSTANT PathCondition     : IN BOOLEAN := TRUE;
      CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
    );

    -- #15
    -- DelayType - VitalDelayType01Z
    -- Input     - Scalar
    -- Output    - Vector
    -- Delay     - Vector
    -- Condition - Vector

    PROCEDURE VitalMemoryAddPathDelay (
      VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
      SIGNAL   InputSignal       : IN STD_ULOGIC;
      CONSTANT OutputSignalName  : IN STRING := "";
      VARIABLE InputChangeTime   : INOUT Time;
      CONSTANT PathDelayArray    : IN VitalDelayArrayType01Z;
      CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
      CONSTANT PathConditionArray: IN VitalBoolArrayT;
      CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
    );

    -- #16
    -- DelayType - VitalDelayType01Z
    -- Input     - Vector
    -- Output    - Scalar
    -- Delay     - Vector
    -- Condition - Scalar

    PROCEDURE VitalMemoryAddPathDelay (
      VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
      SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
      CONSTANT OutputSignalName     : IN STRING := "";
      VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
      CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
      CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
      CONSTANT PathCondition        : IN BOOLEAN := TRUE;
      CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
      CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
    );

    -- #17
    -- DelayType - VitalDelayType01Z
    -- Input     - Vector
    -- Output    - Vector
```

Copyright © 2001 IEEE. All rights reserved.

```
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
);

-- #18
-- DelayType - VitalDelayType01Z
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
);

-- #19
-- DelayType - VitalDelayType01ZX
-- Input     - Scalar
-- Output    - Scalar
-- Delay     - Scalar
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_ULOGIC;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTime      : INOUT Time;
  CONSTANT PathDelay            : IN VitalDelayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE
);

-- #20
-- DelayType - VitalDelayType01ZX
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
);

-- #21
-- DelayType - VitalDelayType01ZX
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
```


Copyright © 2001 IEEE. All rights reserved.

```
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
);

-- #22
-- DelayType - VitalDelayType01ZX
-- Input     - Vector
-- Output    - Scalar
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
);

-- #23
-- DelayType - VitalDelayType01ZX
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
);

-- #24
-- DelayType - VitalDelayType01ZX
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector

PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
);

-- ----------------------------------------------------------------------------
--
-- Function     :   VitalMemorySchedulePathDelay
--
-- Arguments:
--
--  OUT                 Type                  Description
--  OutSignal           STD_LOGIC_VECTOR/     The output signal for
--                      STD_ULOGIC            scheduling
--
--  IN
--  OutputSignalName    STRING                The name of the output signal
--
--  IN
--  PortFlag            VitalPortFlagType     Port flag variable from
--                                            functional procedures
--
```


Copyright © 2001 IEEE. All rights reserved.

```
--  IN
--  OutputMap          VitalOutputMapType   For VitalPathDelay01Z, the
--                                          output can be mapped to
--                                          alternate strengths to model
--                                          tri-state devices, pull-ups
--                                          and pull-downs.
--
--  INOUT
--  ScheduleDataArray/ VitalMemoryScheduleDataVectorType/
--  ScheduleData       VitalMemoryScheduleDataType
--                                          Internal data variable for
--                                          storing delay and schedule
--                                          information for each
--                                          output bit
--
-- ----------------------------------------------------------------------------
--
-- ScheduleDataArray - Vector
-- OutputSignal - Vector
--
PROCEDURE VitalMemorySchedulePathDelay (
  SIGNAL   OutSignal         : OUT std_logic_vector;
  CONSTANT OutputSignalName  : IN STRING := "";
  CONSTANT PortFlag          : IN VitalPortFlagType := VitalDefaultPortFlag;
  CONSTANT OutputMap : IN VitalOutputMapType := VitalDefaultOutputMap;
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType
);
--
-- ScheduleDataArray - Vector
-- OutputSignal - Vector
--
PROCEDURE VitalMemorySchedulePathDelay (
  SIGNAL   OutSignal         : OUT std_logic_vector;
  CONSTANT OutputSignalName  : IN STRING := "";
  CONSTANT PortFlag          : IN VitalPortFlagVectorType;
  CONSTANT OutputMap : IN VitalOutputMapType := VitalDefaultOutputMap;
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType
);
--
-- ScheduleDataArray - Scalar
-- OutputSignal - Scalar
--
PROCEDURE VitalMemorySchedulePathDelay (
  SIGNAL   OutSignal         : OUT std_ulogic;
  CONSTANT OutputSignalName  : IN STRING := "";
  CONSTANT PortFlag          : IN VitalPortFlagType := VitalDefaultPortFlag;
  CONSTANT OutputMap  : IN VitalOutputMapType := VitalDefaultOutputMap;
  VARIABLE ScheduleData      : INOUT VitalMemoryScheduleDataType
);

-- ----------------------------------------------------------------------------
FUNCTION VitalMemoryTimingDataInit RETURN VitalMemoryTimingDataType;

-- ----------------------------------------------------------------------------
--
-- Function Name:  VitalMemorySetupHoldCheck
--
-- Description:    The VitalMemorySetupHoldCheck procedure detects a setup or a
--                 hold violation on the input test signal with respect
--                 to the corresponding input reference signal.  The timing
--                 constraints are specified through parameters
--                 representing the high and low values for the setup and
--                 hold values for the setup and hold times.  This
--                 procedure assumes non-negative values for setup and hold
--                 timing constraints.
--
--                 It is assumed that negative timing constraints
--                 are handled by internally delaying the test or
--                 reference signals.  Negative setup times result in
--                 a delayed reference signal.  Negative hold times
--                 result in a delayed test signal.  Furthermore, the
--                 delays and constraints associated with these and
--                 other signals may need to be appropriately
--                 adjusted so that all constraint intervals overlap
--                 the delayed reference signals and all constraint
--                 values (with respect to the delayed signals) are
--                 non-negative.
--
--                 This function is overloaded based on the input
--                 TestSignal and reference signals. Parallel, Subword and
--                 Cross Arc relationships between test and reference
```

 Copyright © 2001 IEEE. All rights reserved.

```
--                 signals are supported.
--
-- TestSignal XXXXXXXXXXXX____________________________XXXXXXXXXXXXXXXXXXXXXX
--            :
--            :        -->|       error region       |<--
--            :
--             _______________________________
-- RefSignal                                  \______________________________
--            :           |                  |       |
--            :           |               -->|       |<-- thold
--            :        -->|     tsetup       |<--
--
-- Arguments:
--
--  IN                 Type                 Description
--   TestSignal        std_logic_vector     Value of test signal
--   TestSignalName    STRING               Name of test signal
--   TestDelay         VitalDelayArrayType Model's internal delay associated
--                                          with TestSignal
--   RefSignal         std_ulogic           Value of reference signal
--                     std_logic_vector
--   RefSignalName     STRING               Name of reference signal
--   RefDelay          TIME                 Model's internal delay associated
--                     VitalDelayArrayType with RefSignal
--   SetupHigh         VitalDelayArrayType Absolute minimum time duration
--                                          before the transition of RefSignal
--                                          for which transitions of
--                                          TestSignal are allowed to proceed
--                                          to the "1" state without causing
--                                          a setup violation.
--   SetupLow          VitalDelayArrayType Absolute minimum time duration
--                                          before the transition of RefSignal
--                                          for which transitions of
--                                          TestSignal are allowed to proceed
--                                          to the "0" state without causing
--                                          a setup violation.
--   HoldHigh          VitalDelayArrayType Absolute minimum time duration
--                                          after the transition of RefSignal
--                                          for which transitions of
--                                          TestSignal are allowed to
--                                          proceed to the "1" state without
--                                          causing a hold violation.
--   HoldLow           VitalDelayArrayType Absolute minimum time duration
--                                          after the transition of RefSignal
--                                          for which transitions of
--                                          TestSignal are allowed to
--                                          proceed to the "0" state without
--                                          causing a hold violation.
--   CheckEnabled      BOOLEAN              Check performed if TRUE.
--   RefTransition     VitalEdgeSymbolType
--                                          Reference edge specified. Events
--                                          on the RefSignal which match the
--                                          edge spec. are used as reference
--                                          edges.
--   ArcType           VitalMemoryArcType
--   NumBitsPerSubWord INTEGER
--   HeaderMsg         STRING               String that will accompany any
--                                          assertion messages produced.
--   XOn               BOOLEAN              If TRUE, Violation output
--                                          parameter is set to "X".
--                                          Otherwise, Violation is always
--                                          set to "0."
--   MsgOn             BOOLEAN              If TRUE, set and hold violation
--                                          message will be generated.
--                                          Otherwise, no messages are
--                                          generated, even upon violations.
--   MsgSeverity       SEVERITY_LEVEL       Severity level for the assertion.
--   MsgFormat         VitalMemoryMsgFormatType
--                                          Format of the Test/Reference
--                                          signals in violation messages.
--
--  INOUT
--   TimingData        VitalMemoryTimingDataType
--                                          VitalMemorySetupHoldCheck information
--                                          storage area.  This is used
--                                          internally to detect reference
--                                          edges and record the time of the
--                                          last edge.
--
--  OUT
--   Violation         X01              This is the violation flag returned.
```

Copyright © 2001 IEEE. All rights reserved.

```
--                    X01ArrayT        Overloaded for array type.
--
--
-- ---------------------------------------------------------------------------

PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_ulogic;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     TIME := 0 ns;
  SIGNAL   RefSignal         : IN     std_ulogic;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     TIME := 0 ns;
  CONSTANT SetupHigh         : IN     VitalDelayType;
  CONSTANT SetupLow          : IN     VitalDelayType;
  CONSTANT HoldHigh          : IN     VitalDelayType;
  CONSTANT HoldLow           : IN     VitalDelayType;
  CONSTANT CheckEnabled      : IN     VitalBoolArrayT;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
);

PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_ulogic;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     TIME := 0 ns;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
);

PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_ulogic;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     TIME := 0 ns;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     VitalBoolArrayT;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
  );

  PROCEDURE VitalMemorySetupHoldCheck (
    VARIABLE Violation         : OUT    X01ArrayT;
    VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
    SIGNAL   TestSignal        : IN     std_logic_vector;
    CONSTANT TestSignalName    : IN     STRING := "";
    CONSTANT TestDelay         : IN     VitalDelayArrayType;
    SIGNAL   RefSignal         : IN     std_logic_vector;
    CONSTANT RefSignalName     : IN     STRING := "";
    CONSTANT RefDelay          : IN     VitalDelayArrayType;
    CONSTANT SetupHigh         : IN     VitalDelayArrayType;
    CONSTANT SetupLow          : IN     VitalDelayArrayType;
    CONSTANT HoldHigh          : IN     VitalDelayArrayType;
    CONSTANT HoldLow           : IN     VitalDelayArrayType;
    CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
    CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
    CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
    CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
    CONSTANT HeaderMsg         : IN     STRING := " ";
    CONSTANT XOn               : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
    CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
    CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
    CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
    CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
  );

  PROCEDURE VitalMemorySetupHoldCheck (
    VARIABLE Violation         : OUT    X01ArrayT;
    VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
    SIGNAL   TestSignal        : IN     std_logic_vector;
    CONSTANT TestSignalName    : IN     STRING := "";
    CONSTANT TestDelay         : IN     VitalDelayArrayType;
    SIGNAL   RefSignal         : IN     std_logic_vector;
    CONSTANT RefSignalName     : IN     STRING := "";
    CONSTANT RefDelay          : IN     VitalDelayArrayType;
    CONSTANT SetupHigh         : IN     VitalDelayArrayType;
    CONSTANT SetupLow          : IN     VitalDelayArrayType;
    CONSTANT HoldHigh          : IN     VitalDelayArrayType;
    CONSTANT HoldLow           : IN     VitalDelayArrayType;
    CONSTANT CheckEnabled      : IN     VitalBoolArrayT;
    CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
    CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
    CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
    CONSTANT HeaderMsg         : IN     STRING := " ";
    CONSTANT XOn               : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
    CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
    CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
    CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
    CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
  );

  --------------- following are not needed --------------------------

  PROCEDURE VitalMemorySetupHoldCheck (
    VARIABLE Violation         : OUT    X01;
    VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
    SIGNAL   TestSignal        : IN     std_logic_vector;
    CONSTANT TestSignalName    : IN     STRING := "";
    CONSTANT TestDelay         : IN     VitalDelayArrayType;
    SIGNAL   RefSignal         : IN     std_ulogic;
    CONSTANT RefSignalName     : IN     STRING := "";
    CONSTANT RefDelay          : IN     TIME := 0 ns;
    CONSTANT SetupHigh         : IN     VitalDelayArrayType;
    CONSTANT SetupLow          : IN     VitalDelayArrayType;
    CONSTANT HoldHigh          : IN     VitalDelayArrayType;
    CONSTANT HoldLow           : IN     VitalDelayArrayType;
    CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
    CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
    CONSTANT HeaderMsg         : IN     STRING := " ";
    CONSTANT XOn               : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
```

Copyright © 2001 IEEE. All rights reserved.

```
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
);

PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_logic_vector;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     VitalDelayArrayType;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
);


-- ------------------------------------------------------------------------------
--
-- Function Name:  VitalPeriodPulseCheck
--
-- Description:    VitalPeriodPulseCheck checks for minimum and maximum
--                 periodicity and pulse width for "1" and "0" values of
--                 the input test signal.  The timing constraint is
--                 specified through parameters representing the minimal
--                 period between successive rising and falling edges of
--                 the input test signal and the minimum pulse widths
--                 associated with high and low values.
--
--                 VitalPeriodCheck's accepts rising and falling edges
--                 from 1 and 0 as well as transitions to and from 'X.'
--
--                    _______________         __________
--       ____________|               |_______|
--
--                   |<--- pw_hi --->|
--                   |<-------- period ----->|
--                                -->| pw_lo |<--
--
-- Arguments:
--    IN             Type               Description
--    TestSignal     std_logic_vector   Value of test signal
--    TestSignalName STRING             Name of the test signal
--    TestDelay      VitalDelayArrayType
--                                      Model's internal delay associated
--                                      with TestSignal
--    Period         VitalDelayArrayType
--                                      Minimum period allowed between
--                                      consecutive rising ('P') or
--                                      falling ('F') transitions.
--    PulseWidthHigh VitalDelayArrayType
--                                      Minimum time allowed for a high
--                                      pulse ('1' or 'H')
--    PulseWidthLow  VitalDelayArrayType
--                                      Minimum time allowed for a low
--                                      pulse ('0' or 'L')
--    CheckEnabled   BOOLEAN            Check performed if TRUE.
--    HeaderMsg      STRING             String that will accompany any
--                                      assertion messages produced.
--    XOn            BOOLEAN            If TRUE, Violation output parameter
--                                      is set to "X". Otherwise, Violation
--                                      is always set to "0."
--    MsgOn          BOOLEAN            If TRUE, period/pulse violation
```

 Copyright © 2001 IEEE. All rights reserved.

```
--                                    message will be generated.
--                                    Otherwise, no messages are generated,
--                                    even though a violation is detected.
--    MsgSeverity    SEVERITY_LEVEL   Severity level for the assertion.
--    MsgFormat      VitalMemoryMsgFormatType
--                                    Format of the Test/Reference signals
--                                    in violation messages.
--
--    INOUT
--    PeriodData     VitalPeriodDataArrayType
--                                    VitalPeriodPulseCheck information
--                                    storage area.  This is used
--                                    internally to detect reference edges
--                                    and record the pulse and period
--                                    times.
--    OUT
--    Violation      X01              This is the violation flag returned.
--                   X01ArrayT        Overloaded for array type.
--
-- ----------------------------------------------------------------------------
PROCEDURE  VitalMemoryPeriodPulseCheck  (
  VARIABLE Violation      : OUT   X01ArrayT;
  VARIABLE PeriodData     : INOUT VitalPeriodDataArrayType;
  SIGNAL   TestSignal     : IN    std_logic_vector;
  CONSTANT TestSignalName : IN    STRING := "";
  CONSTANT TestDelay      : IN    VitalDelayArrayType;
  CONSTANT Period         : IN    VitalDelayArrayType;
  CONSTANT PulseWidthHigh : IN    VitalDelayArrayType;
  CONSTANT PulseWidthLow  : IN    VitalDelayArrayType;
  CONSTANT CheckEnabled   : IN    BOOLEAN := TRUE;
  CONSTANT HeaderMsg      : IN    STRING := " ";
  CONSTANT XOn            : IN    BOOLEAN := TRUE;
  CONSTANT MsgOn          : IN    BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN    SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat      : IN    VitalMemoryMsgFormatType
);

PROCEDURE  VitalMemoryPeriodPulseCheck  (
  VARIABLE Violation      : OUT   X01;
  VARIABLE PeriodData     : INOUT VitalPeriodDataArrayType;
  SIGNAL   TestSignal     : IN    std_logic_vector;
  CONSTANT TestSignalName : IN    STRING := "";
  CONSTANT TestDelay      : IN    VitalDelayArrayType;
  CONSTANT Period         : IN    VitalDelayArrayType;
  CONSTANT PulseWidthHigh : IN    VitalDelayArrayType;
  CONSTANT PulseWidthLow  : IN    VitalDelayArrayType;
  CONSTANT CheckEnabled   : IN    BOOLEAN := TRUE;
  CONSTANT HeaderMsg      : IN    STRING := " ";
  CONSTANT XOn            : IN    BOOLEAN := TRUE;
  CONSTANT MsgOn          : IN    BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN    SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat      : IN    VitalMemoryMsgFormatType
);

-- ----------------------------------------------------------------------------
-- Functionality Section
-- ----------------------------------------------------------------------------

-- ----------------------------------------------------------------------------
-- All Memory Types and Record definitions.
-- ----------------------------------------------------------------------------
TYPE MemoryWordType        IS ARRAY (NATURAL RANGE <>) OF UX01;
TYPE MemoryWordPtr         IS ACCESS MemoryWordType;

TYPE MemoryArrayType       IS ARRAY (NATURAL RANGE <>) OF MemoryWordPtr;
TYPE MemoryArrayPtrType    IS ACCESS MemoryArrayType;

TYPE VitalMemoryArrayRecType IS
RECORD
NoOfWords           : POSITIVE;
NoOfBitsPerWord     : POSITIVE;
NoOfBitsPerSubWord  : POSITIVE;
NoOfBitsPerEnable   : POSITIVE;
MemoryArrayPtr      : MemoryArrayPtrType;
END RECORD;

TYPE VitalMemoryDataType    IS ACCESS VitalMemoryArrayRecType;

TYPE VitalTimingDataVectorType IS
ARRAY (NATURAL RANGE <>) OF VitalTimingDataType;
```

Copyright © 2001 IEEE. All rights reserved.

```
TYPE VitalMemoryViolFlagSizeType IS ARRAY (NATURAL RANGE <>) OF INTEGER;

-- ----------------------------------------------------------------------------
-- Symbol Literals used for Memory Table Modeling
-- ----------------------------------------------------------------------------

-- Symbol literals from '/' to 'S' are closely related to MemoryTableMatch
-- lookup matching and the order cannot be arbitrarily changed.
-- The remaining symbol literals are interpreted directly and matchting is
-- handled in the MemoryMatch procedure itself.

TYPE VitalMemorySymbolType IS (
  '/',      -- 0 -> 1
  '\',      -- 1 -> 0
  'P',      -- Union of '/' and '^' (any edge to 1)
  'N',      -- Union of '\' and 'v' (any edge to 0)
  'r',      -- 0 -> X
  'f',      -- 1 -> X
  'p',      -- Union of '/' and 'r' (any edge from 0)
  'n',      -- Union of '\' and 'f' (any edge from 1)
  'R',      -- Union of '^' and 'p' (any possible rising edge)
  'F',      -- Union of 'v' and 'n' (any possible falling edge)
  '^',      -- X -> 1
  'v',      -- X -> 0
  'E',      -- Union of 'v' and '^' (any edge from X)
  'A',      -- Union of 'r' and '^' (rising edge to or from 'X')

  'D',      -- Union of 'f' and 'v' (falling edge to or from 'X')

  '*',      -- Union of 'R' and 'F' (any edge)
  'X',      -- Unknown level
  '0',      -- low level
  '1',      -- high level
  '-',      -- don't care
  'B',      -- 0 or 1
  'Z',      -- High Impedance
  'S',      -- steady value

  'g',      -- Good address (no transition)
  'u',      -- Unknown address (no transition)
  'i',      -- Invalid address (no transition)
  'G',      -- Good address (with transition)
  'U',      -- Unknown address (with transition)
  'I',      -- Invalid address (with transition)

  'w',      -- Write data to memory
  's',      -- Retain previous memory contents

  'c',      -- Corrupt entire memory with 'X'
  'l',      -- Corrupt a word in memory with 'X'
  'd',      -- Corrupt a single bit in memory with 'X'
  'e',      -- Corrupt a word with 'X' based on data in
  'C',      -- Corrupt a sub-word entire memory with 'X'
  'L',      -- Corrupt a sub-word in memory with 'X'

  -- The following entries are commented since their
  -- interpretation overlap with existing definitions.

  --      'D',      -- Corrupt a single bit of a sub-word with 'X'
  --      'E',      -- Corrupt a sub-word with 'X' based on datain

  'M',      -- Implicit read data from memory
  'm',      -- Read data from memory
  't'       -- Immediate assign/transfer data in

);

TYPE VitalMemoryTableType IS ARRAY ( NATURAL RANGE <>, NATURAL RANGE <> )
  OF VitalMemorySymbolType;

TYPE VitalMemoryViolationSymbolType IS (
  'X',      -- Unknown level
  '0',      -- low level
  '-'       -- don't care
);

TYPE VitalMemoryViolationTableType IS
  ARRAY ( NATURAL RANGE <>, NATURAL RANGE <> )
  OF VitalMemoryViolationSymbolType;

TYPE VitalPortType IS (
```

 Copyright © 2001 IEEE. All rights reserved.

```
    UNDEF,
    READ,
    WRITE,
    RDNWR
  );

  TYPE VitalCrossPortModeType IS (
    CpRead,                          --  CpReadOnly,
    WriteContention,                 --  WrContOnly,
    ReadWriteContention,             --  CpContention
    CpReadAndWriteContention,        --  WrContAndCpRead,
    CpReadAndReadContention
  );

  SUBTYPE VitalAddressValueType IS INTEGER;
  TYPE VitalAddressValueVectorType IS
    ARRAY (NATURAL RANGE <>) OF VitalAddressValueType;

  -- ----------------------------------------------------------------------------
  -- Procedure:   VitalDeclareMemory
  -- Parameters:  NoOfWords          - Number of words in the memory
  --              NoOfBitsPerWord    - Number of bits per word in memory
  --              NoOfBitsPerSubWord - Number of bits per sub word
  --              MemoryLoadFile     - Name of data file to load
  -- Description: This function is intended to be used to initialize
  --              memory data declarations, i.e. to be executed duing
  --              simulation elaboration time.  Handles the allocation
  --              and initialization of memory for the memory data.
  --              Default NoOfBitsPerSubWord is NoOfBits.
  -- ----------------------------------------------------------------------------

  FUNCTION VitalDeclareMemory (
    CONSTANT NoOfWords          : IN POSITIVE;
    CONSTANT NoOfBitsPerWord    : IN POSITIVE;
    CONSTANT NoOfBitsPerSubWord : IN POSITIVE;
    CONSTANT MemoryLoadFile     : IN string := "";
    CONSTANT BinaryLoadFile     : IN BOOLEAN := FALSE
  ) RETURN VitalMemoryDataType;

  FUNCTION VitalDeclareMemory (
    CONSTANT NoOfWords          : IN POSITIVE;
    CONSTANT NoOfBitsPerWord    : IN POSITIVE;
    CONSTANT MemoryLoadFile     : IN string := "";
    CONSTANT BinaryLoadFile     : IN BOOLEAN := FALSE
  ) RETURN VitalMemoryDataType;


  -- ----------------------------------------------------------------------------
  -- Procedure:   VitalMemoryTable
  -- Parameters:  DataOutBus     - Output candidate zero delay data bus out
  --              MemoryData     - Pointer to memory data structure
  --              PrevControls   - Previous data in for edge detection
  --              PrevEnableBus  - Previous enables for edge detection
  --              PrevDataInBus  - Previous data bus for edge detection
  --              PrevAddressBus - Previous address bus for edge detection
  --              PortFlag       - Indicates port operating mode
  --              PortFlagArray  - Vector form of PortFlag for sub-word
  --              Controls       - Agregate of scalar control lines
  --              EnableBus      - Concatenation of vector control lines
  --              DataInBus      - Input value of data bus in
  --              AddressBus     - Input value of address bus in
  --              AddressValue   - Decoded value of the AddressBus
  --              MemoryTable    - Input memory action table
  --              PortType       - The type of port (currently not used)
  --              PortName       - Port name string for messages
  --              HeaderMsg      - Header string for messages
  --              MsgOn          - Control the generation of messages
  --              MsgSeverity    - Control level of message generation
  -- Description: This procedure implements the majority of the memory
  --              modeling functionality via lookup of the memory action
  --              tables and performing the specified actions if matches
  --              are found, or the default actions otherwise.  The
  --              overloadings are provided for the word and sub-word
  --              (using the EnableBus and PortFlagArray arguments) addressing
  --              cases.
  -- ----------------------------------------------------------------------------

  PROCEDURE VitalMemoryTable (
    VARIABLE DataOutBus     : INOUT std_logic_vector;
    VARIABLE MemoryData     : INOUT VitalMemoryDataType;
    VARIABLE PrevControls   : INOUT std_logic_vector;
```

Copyright © 2001 IEEE. All rights reserved.

```
      VARIABLE PrevDataInBus  : INOUT std_logic_vector;
      VARIABLE PrevAddressBus : INOUT std_logic_vector;
      VARIABLE PortFlag       : INOUT VitalPortFlagVectorType;
      CONSTANT Controls       : IN std_logic_vector;
      CONSTANT DataInBus      : IN std_logic_vector;
      CONSTANT AddressBus     : IN std_logic_vector;
      VARIABLE AddressValue   : INOUT VitalAddressValueType;
      CONSTANT MemoryTable    : IN VitalMemoryTableType;
      CONSTANT PortType       : IN VitalPortType := UNDEF;
      CONSTANT PortName       : IN STRING := "";
      CONSTANT HeaderMsg      : IN STRING := "";
      CONSTANT MsgOn          : IN BOOLEAN := TRUE;
      CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
    );

    PROCEDURE VitalMemoryTable (
      VARIABLE DataOutBus     : INOUT std_logic_vector;
      VARIABLE MemoryData     : INOUT VitalMemoryDataType;
      VARIABLE PrevControls   : INOUT std_logic_vector;
      VARIABLE PrevEnableBus  : INOUT std_logic_vector;
      VARIABLE PrevDataInBus  : INOUT std_logic_vector;
      VARIABLE PrevAddressBus : INOUT std_logic_vector;
      VARIABLE PortFlagArray  : INOUT VitalPortFlagVectorType;
      CONSTANT Controls       : IN std_logic_vector;
      CONSTANT EnableBus      : IN std_logic_vector;
      CONSTANT DataInBus      : IN std_logic_vector;
      CONSTANT AddressBus     : IN std_logic_vector;
      VARIABLE AddressValue   : INOUT VitalAddressValueType;
      CONSTANT MemoryTable    : IN VitalMemoryTableType;
      CONSTANT PortType       : IN VitalPortType := UNDEF;
      CONSTANT PortName       : IN STRING := "";
      CONSTANT HeaderMsg      : IN STRING := "";
      CONSTANT MsgOn          : IN BOOLEAN := TRUE;
      CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
    );

    -- ------------------------------------------------------------------------------
    -- Procedure:   VitalMemoryCrossPorts
    -- Parameters:  DataOutBus     - Output candidate zero delay data bus out
    --              MemoryData     - Pointer to memory data structure
    --              SamePortFlag   - Operating mode for same port
    --              SamePortAddressValue  - Decoded AddressBus for same port
    --              CrossPortFlagArray    - Operating modes for cross ports
    --              CrossPortAddressArray - Decoded AddressBus for cross ports
    --              CrossPortMode  - Write contention and crossport read control
    --              PortName       - Port name string for messages
    --              HeaderMsg      - Header string for messages
    --              MsgOn          - Control the generation of messages
    --
    -- Description: These procedures control the effect of memory operations
    --              on a given port due to operations on other ports in a
    --              multi-port memory.
    --              This includes data write through when reading and writing
    --              to the same address, as well as write contention when
    --              there are multiple write to the same address.
    --              If addresses do not match then data bus is unchanged.
    --              The DataOutBus can be diabled with 'Z' value.
    -- ------------------------------------------------------------------------------

    PROCEDURE VitalMemoryCrossPorts (
      VARIABLE DataOutBus            : INOUT std_logic_vector;
      VARIABLE MemoryData            : INOUT VitalMemoryDataType;
      VARIABLE SamePortFlag          : INOUT VitalPortFlagVectorType;
      CONSTANT SamePortAddressValue  : IN VitalAddressValueType;
      CONSTANT CrossPortFlagArray    : IN VitalPortFlagVectorType;
      CONSTANT CrossPortAddressArray : IN VitalAddressValueVectorType;
      CONSTANT CrossPortMode         : IN VitalCrossPortModeType
                                           := CpReadAndWriteContention;
      CONSTANT PortName              : IN STRING := "";
      CONSTANT HeaderMsg             : IN STRING := "";
      CONSTANT MsgOn                 : IN BOOLEAN := TRUE
    ) ;

    PROCEDURE VitalMemoryCrossPorts (
      VARIABLE MemoryData            : INOUT VitalMemoryDataType;
      CONSTANT CrossPortFlagArray    : IN VitalPortFlagVectorType;
      CONSTANT CrossPortAddressArray : IN VitalAddressValueVectorType;
      CONSTANT HeaderMsg             : IN STRING := "";
      CONSTANT MsgOn                 : IN BOOLEAN := TRUE
    ) ;
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- ----------------------------------------------------------------------------
-- Procedure:   VitalMemoryViolation
-- Parameters:  DataOutBus          - Output zero delay data bus out
--              MemoryData          - Pointer to memory data structure
--              PortFlag            - Indicates port operating mode
--              DataInBus           - Input value of data bus in
--              AddressValue        - Decoded value of the AddressBus
--              ViolationFlags      - Aggregate of scalar violation vars
--              ViolationFlagsArray - Concatenation of vector violation vars
--              ViolationTable      - Input memory violation table
--              PortType            - The type of port (currently not used)
--              PortName            - Port name string for messages
--              HeaderMsg           - Header string for messages
--              MsgOn               - Control the generation of messages
--              MsgSeverity         - Control level of message generation
-- Description: This procedure is intended to implement all actions on the
--              memory contents and data out bus as a result of timing viols.
--              It uses the memory action table to perform various corruption
--              policies specified by the user.
-- ----------------------------------------------------------------------------

PROCEDURE VitalMemoryViolation (
  VARIABLE DataOutBus           : INOUT std_logic_vector;
  VARIABLE MemoryData           : INOUT VitalMemoryDataType;
  VARIABLE PortFlag             : INOUT VitalPortFlagVectorType;
  CONSTANT DataInBus            : IN std_logic_vector;
  CONSTANT AddressValue         : IN VitalAddressValueType;
  CONSTANT ViolationFlags       : IN std_logic_vector;
  CONSTANT ViolationFlagsArray  : IN X01ArrayT;
  CONSTANT ViolationSizesArray  : IN VitalMemoryViolFlagSizeType;
  CONSTANT ViolationTable       : IN VitalMemoryTableType;
  CONSTANT PortType             : IN VitalPortType;
  CONSTANT PortName             : IN STRING := "";
  CONSTANT HeaderMsg            : IN STRING := "";
  CONSTANT MsgOn                : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity          : IN SEVERITY_LEVEL := WARNING
) ;

PROCEDURE VitalMemoryViolation (
  VARIABLE DataOutBus           : INOUT std_logic_vector;
  VARIABLE MemoryData           : INOUT VitalMemoryDataType;
  VARIABLE PortFlag             : INOUT VitalPortFlagVectorType;
  CONSTANT DataInBus            : IN std_logic_vector;
  CONSTANT AddressValue         : IN VitalAddressValueType;
  CONSTANT ViolationFlags       : IN std_logic_vector;
  CONSTANT ViolationTable       : IN VitalMemoryTableType;
  CONSTANT PortType             : IN VitalPortType;
  CONSTANT PortName             : IN STRING := "";
  CONSTANT HeaderMsg            : IN STRING := "";
  CONSTANT MsgOn                : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity          : IN SEVERITY_LEVEL := WARNING
) ;

END Vital_Memory;
```

13.6 VITAL_Memory package body

```
-- ----------------------------------------------------------------------------
-- Title         : Standard VITAL Memory Package
--               :
-- Library       : Vital_Memory
--               :
-- Developers    : IEEE DASC Timing Working Group (TWG), PAR 1076.4
--               : Ekambaram Balaji, LSI Logic Corporation
--               : Jose De Castro, Consultant
--               : Prakash Bare, GDA Technologies
--               : William Yam, LSI Logic Corporation
--               : Dennis Brophy, Model Technology
--               :
-- Purpose       : This packages defines standard types, constants, functions
--               : and procedures for use in developing ASIC memory models.
--               :
-- ----------------------------------------------------------------------------
--
-- ----------------------------------------------------------------------------
-- Modification History :
-- ----------------------------------------------------------------------------
-- Ver:|Auth:| Date:| Changes Made:
-- 0.1 | eb  |071796| First prototye as part of VITAL memory proposal
```

Copyright © 2001 IEEE. All rights reserved.

```
-- 0.2 | jdc |012897| Initial prototyping with proposed MTM scheme
-- 0.3 | jdc |090297| Extensive updates for TAG review (functional)
-- 0.4 | eb  |091597| Changed naming conventions for VitalMemoryTable
--     |     |      | Added interface of VitalMemoryCrossPorts() &
--     |     |      | VitalMemoryViolation().
-- 0.5 | jdc |092997| Completed naming changes thoughout package body.
--     |     |      | Testing with simgle port test model looks ok.
-- 0.6 | jdc |121797| Major updates to the packages:
--     |     |      | - Implement VitalMemoryCrossPorts()
--     |     |      |   - Use new VitalAddressValueType
--     |     |      |   - Use new VitalCrossPortModeType enum
--     |     |      |   - Overloading without SamePort args
--     |     |      |   - Honor erroneous address values
--     |     |      |   - Honor ports disabled with 'Z'
--     |     |      | - Implement implicit read 'M' table symbol
--     |     |      | - Cleanup buses to use (H DOWNTO L)
--     |     |      | - Message control via MsgOn,HeaderMsg,PortName
--     |     |      | - Tested with 1P1RW,2P2RW,4P2R2W,4P4RW cases
-- 0.7 | jdc |052698| Bug fixes to the packages:
--     |     |      | - Fix failure with negative Address values
--     |     |      | - Added debug messages for VMT table search
--     |     |      | - Remove 'S' for action column (only 's')
--     |     |      | - Remove 's' for response column (only 'S')
--     |     |      | - Remove 'X' for action and response columns
-- 0.8 | jdc |061298| Implemented VitalMemoryViolation()
--     |     |      | - Minimal functionality violation tables
--     |     |      | - Missing:
--     |     |      |   - Cannot handle wide violation variables
--     |     |      |   - Cannot handle sub-word cases
--     |     |      | Fixed IIC version of MemoryMatch
--     |     |      | Fixed 'M' vs 'm' switched on debug output
--     |     |      | TO BE DONE:
--     |     |      | - Implement 'd' corrupting a single bit
--     |     |      | - Implement 'D' corrupting a single bit
-- 0.9 |eb/sc|080498| Added UNDEF value for VitalPortFlagType
-- 0.10|eb/sc|080798| Added CORRUPT value for VitalPortFlagType
-- 0.11|eb/sc|081798| Added overloaded function interface for
--     |     |      |       VitalDeclareMemory
-- 0.14| jdc |113198| Merging of memory functionality and version
--     |     |      | 1.4 9/17/98 of timing package from Prakash
-- 0.15| jdc |120198| Major development of VMV functionality
-- 0.16| jdc |120298| Complete VMV functionlality for initial testing
--     |     |      | - New ViolationTableCorruptMask() procedure
--     |     |      | - New MemoryTableCorruptMask() procedure
--     |     |      | - HandleMemoryAction():
--     |     |      |   - Removed DataOutBus bogus output
--     |     |      |   - Replaced DataOutTmp with DataInTmp
--     |     |      |   - Added CorruptMask input handling
--     |     |      |   - Implemented 'd','D' using CorruptMask
--     |     |      |   - CorruptMask on 'd','C','L','D','E'
--     |     |      |   - CorruptMask ignored on 'c','l','e'
--     |     |      |   - Changed 'l','d','e' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'L','D','E' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'c','l','d','e' to ignore HighBit, LowBit
--     |     |      |   - Changed 'C','L','D','E' to use HighBit, LowBit
--     |     |      | - HandleDataAction():
--     |     |      |   - Added CorruptMask input handling
--     |     |      |   - Implemented 'd','D' using CorruptMask
--     |     |      |   - CorruptMask on 'd','C','L','D','E'
--     |     |      |   - CorruptMask ignored on 'l','e'
--     |     |      |   - Changed 'l','d','e' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'L','D','E' to set PortFlag to CORRUPT
--     |     |      |   - Changed 'l','d','e' to ignore HighBit, LowBit
--     |     |      |   - Changed 'L','D','E' to use HighBit, LowBit
--     |     |      | - MemoryTableLookUp():
--     |     |      |   - Added MsgOn table debug output
--     |     |      |   - Uses new MemoryTableCorruptMask()
--     |     |      | - ViolationTableLookUp():
--     |     |      |   - Uses new ViolationTableCorruptMask()
-- 0.17| jdc |120898| - Added VitalMemoryViolationSymbolType,
--     |     |      |   VitalMemoryViolationTableType data
--     |     |      |   types but not used yet (need to discuss)
--     |     |      | - Added overload for VitalMemoryViolation()
--     |     |      |   which does not have array flags
--     |     |      | - Bug fixes for VMV functionality:
--     |     |      |   - ViolationTableLookUp() not handling '-' in
--     |     |      |     scalar violation matching
--     |     |      |   - VitalMemoryViolation() now normalizes
--     |     |      |     VFlagArrayTmp'LEFT as LSB before calling
--     |     |      |     ViolationTableLookUp() for proper scanning
--     |     |      |   - ViolationTableCorruptMask() had to remove
```

 Copyright © 2001 IEEE. All rights reserved.

```
--      |      |       |     normalization of CorruptMaskTmp and
--      |      |       |     ViolMaskTmp for proper MSB:LSB corruption
--      |      |       |   - HandleMemoryAction(), HandleDataAction()
--      |      |       |     - Removed 'D','E' since not being used
--      |      |       |     - Use XOR instead of OR for corrupt masks
--      |      |       |     - Now 'd' is sensitive to HighBit, LowBit
--      |      |       |   - Fixed LowBit overflow in bit writeable case
--      |      |       |     - MemoryTableCorruptMask()
--      |      |       |     - ViolationTableCorruptMask()
--      |      |       |     - VitalMemoryTable()
--      |      |       |     - VitalMemoryCrossPorts()
--      |      |       |   - Fixed VitalMemoryViolation() failing on
--      |      |       |     error AddressValue from earlier VMT()
--      |      |       | - Minor cleanup of code formatting
-- 0.18| jdc  |032599| - In VitalDeclareMemory()
--      |      |       |   - Added BinaryLoadFile formal arg and
--      |      |       |     modified LoadMemory() to handle bin
--      |      |       | - Added NOCHANGE to VitalPortFlagType
--      |      |       | - For VitalCrossPortModeType
--      |      |       |   - Added CpContention enum
--      |      |       | - In HandleDataAction()
--      |      |       |   - Set PortFlag := NOCHANGE for 'S'
--      |      |       | - In HandleMemoryAction()
--      |      |       |   - Set PortFlag := NOCHANGE for 's'
--      |      |       | - In VitalMemoryTable() and
--      |      |       |   VitalMemoryViolation()
--      |      |       |   - Honor PortFlag = NOCHANGE returned
--      |      |       |     from HandleMemoryAction()
--      |      |       | - In VitalMemoryCrossPorts()
--      |      |       |   - Fixed Address = AddressJ for all
--      |      |       |     conditions of DoWrCont & DoCpRead
--      |      |       |   - Handle CpContention like WrContOnly
--      |      |       |     under CpReadOnly conditions, with
--      |      |       |     associated memory message changes
--      |      |       |   - Handle PortFlag = NOCHANGE like
--      |      |       |     PortFlag = READ for actions
--      |      |       | - Modeling change:
--      |      |       |   - Need to init PortFlag every delta
--      |      |       |     PortFlag_A := (OTHES => UNDEF);
--      |      |       | - Updated InternalTimingCheck code
-- 0.19| jdc  |042599| - Fixes for bit-writeable cases
--      |      |       | - Check PortFlag after HandleDataAction
--      |      |       |   in VitalMemoryViolation()
-- 0.20| jdc  |042599| - Merge PortFlag changes from Prakash
--      |      |       |   and Willian:
--      |      |       |     VitalMemorySchedulePathDelay()
--      |      |       |     VitalMemoryExpandPortFlag()
-- 0.21| jdc  |072199| - Changed VitalCrossPortModeType enums,
--      |      |       |   added new CpReadAndReadContention.
--      |      |       | - Fixed VitalMemoryCrossPorts() parameter
--      |      |       |   SamePortFlag to INOUT so that it can
--      |      |       |   set CORRUPT or READ value.
--      |      |       | - Fixed VitalMemoryTable() where PortFlag
--      |      |       |   setting by HandleDataAction() is being
--      |      |       |   ignored when HandleMemoryAction() sets
--      |      |       |   PortFlagTmp to NOCHANGE.
--      |      |       | - Fixed VitalMemoryViolation() to set
--      |      |       |   all bits of PortFlag when violating.
-- 0.22| jdc  |072399| - Added HIGHZ to PortFlagType. HandleData
--      |      |       |   checks whether the previous state is HIGHZ.
--      |      |       |   If yes then portFlag should be NOCHANGE
--      |      |       |   for VMPD to ignore IORetain corruption.
--      |      |       |   The idea is that the first Z should be
--      |      |       |   propagated but later ones should be ignored.
--      |      |       |
-- 0.23| jdc  |100499| - Took code checked in by Dennis 09/28/99
--      |      |       | - Changed VitalPortFlagType to record of
--      |      |       |   new VitalPortStateType to hold current,
--      |      |       |   previous values and separate disable.
--      |      |       |   Also created VitalDefaultPortFlag const.
--      |      |       |   Removed usage of PortFlag NOCHANGE
--      |      |       | - VitalMemoryTable() changes:
--      |      |       |   Optimized return when all curr = prev
--      |      |       |   AddressValue is now INOUT to optimize
--      |      |       |   Transfer PF.MemoryCurrent to MemoryPrevious
--      |      |       |   Transfer PF.DataCurrent to DataPrevious
--      |      |       |   Reset PF.OutputDisable to FALSE
--      |      |       |   Expects PortFlag init in declaration
--      |      |       |   No need to init PortFlag every delta
--      |      |       | - VitalMemorySchedulePathDelay() changes:
--      |      |       |   Initialize with VitalDefaultPortFlag
```

Copyright © 2001 IEEE. All rights reserved.

```
--      |      |      |   Check PortFlag.OutputDisable
--      |      |      | - HandleMemoryAction() changes:
--      |      |      |   Set value of PortFlag.MemoryCurrent
--      |      |      |   Never set PortFlag.OutputDisable
--      |      |      | - HandleDataAction() changes:
--      |      |      |   Set value of PortFlag.DataCurrent
--      |      |      |   Set PortFlag.DataCurrent for HIGHZ
--      |      |      | - VitalMemoryCrossPorts() changes:
--      |      |      |   Check/set value of PF.MemoryCurrent
--      |      |      |   Check value of PF.OutputDisable
--      |      |      | - VitalMemoryViolation() changes:
--      |      |      |   Fixed bug - not reading inout PF value
--      |      |      |   Clean up setting of PortFlag
-- 0.24| jdc |100899| - Modified update of PF.OutputDisable
--      |      |      |   to correctly accomodate 2P1W1R case:
--      |      |      |   the read port should not exhibit
--      |      |      |   IO retain corrupt when reading
--      |      |      |   addr unrelated to addr being written.
-- 0.25| jdc |100999| - VitalMemoryViolation() change:
--      |      |      |   Fixed bug with RDNWR mode incorrectly
--      |      |      |   updating the PF.OutputDisable
-- 0.26| jdc |100999| - VitalMemoryCrossPorts() change:
--      |      |      |   Fixed bugs with update of PF
-- 0.27| jdc |101499| - VitalMemoryCrossPorts() change:
--      |      |      |   Added DoRdWrCont message (ErrMcpRdWrCo,
--      |      |      |   Memory cross port read/write data only
--      |      |      |   contention)
--      |      |      | - VitalMemoryTable() change:
--      |      |      |   Set PF.OutputDisable := TRUE for the
--      |      |      |   optimized cases.
-- 0.28| pb  |112399| - Added 8 VMPD procedures for vector
--      |      |      |   PathCondition support. Now the total
--      |      |      |   number of overloadings for VMPD is 24.
--      |      |      | - Number of overloadings for SetupHold
--      |      |      |   procedures increased to 5. Scalar violations
--      |      |      |   are not supported anymore. Vector checkEnabled
--      |      |      |   support is provided through the new overloading
-- 0.29| jdc |120999| - HandleMemoryAction() HandleDataAction()
--      |      |      |   Reinstated 'D' and 'E' actions but
--      |      |      |   with new PortFlagType
--      |      |      | - Updated file handling syntax, must compile
--      |      |      |   with -93 syntax now.
-- 0.30| jdc |022300| - Formated for 80 column max width
-- ----------------------------------------------------------------------------

LIBRARY IEEE;
USE     IEEE.STD_LOGIC_1164.ALL;
USE     IEEE.Vital_Timing.all;
USE     IEEE.Vital_Primitives.all;

LIBRARY STD;
USE     STD.TEXTIO.ALL;

-- ----------------------------------------------------------------------------
PACKAGE BODY Vital_Memory IS

-- ----------------------------------------------------------------------------
-- Timing Section
-- ----------------------------------------------------------------------------

FILE LogFile : TEXT OPEN write_mode IS "delayLog";
FILE Output  : TEXT OPEN write_mode IS "STD_OUTPUT";

-- Added for turning off the debug msg..
CONSTANT PrintDebugMsg : STD_ULOGIC := '0';
                         -- '0' - don't print in STD OUTPUT
                         -- '1' - print in STD OUTPUT

-- Type and constant definitions for type conversion.
TYPE MVL9_TO_CHAR_TBL IS ARRAY (STD_ULOGIC) OF character;

--constant MVL9_to_char: MVL9_TO_CHAR_TBL := "UX01ZWLH-";
CONSTANT MVL9_to_char: MVL9_TO_CHAR_TBL := "XX01ZX010";

-- ----------------------------------------------------------------------------
-- STD_LOGIC WRITE UTILITIES
-- ----------------------------------------------------------------------------
PROCEDURE WRITE(
  l       : INOUT line;
  val     : IN std_logic_vector;
  justify : IN side := right;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    field   : IN width := 0
  ) IS
    VARIABLE invect : std_logic_vector(val'LENGTH DOWNTO 1);
    VARIABLE ins    : STRING(val'LENGTH DOWNTO 1);
  BEGIN
    invect := val;
    FOR I IN invect'length DOWNTO 1 LOOP
       ins(I) := MVL9_to_char(invect(I));
    END LOOP;
    WRITE(L, ins, justify, field);
  END;

  PROCEDURE WRITE(
    l       : INOUT line;
    val     : IN std_ulogic;
    justify : IN side := right;
    field   : in width := 0
  ) IS
    VARIABLE ins : CHARACTER;
  BEGIN
    ins := MVL9_to_char(val);
    WRITE(L, ins, justify, field);
  END;

  -- ------------------------------------------------------------------------------
  PROCEDURE DelayValue(
    InputTime : IN TIME ;
    outline   : INOUT LINE
  ) IS
    CONSTANT header : STRING := "TIME'HIGH";
  BEGIN
    IF(InputTime = TIME'HIGH) THEN
      WRITE(outline, header);
    ELSE
      WRITE(outline, InputTime);
    END IF;
  END DelayValue;

  -- ------------------------------------------------------------------------------
  PROCEDURE PrintScheduleDataArray (
    ScheduleDataArray : IN VitalMemoryScheduleDataVectorType
  ) IS
    VARIABLE outline1 : LINE;
    VARIABLE outline2 : LINE;
    VARIABLE value    : TIME;
    CONSTANT empty    : STRING := "      ";
    CONSTANT header1  : STRING := "i    Age     PropDly  RetainDly";
    CONSTANT header2  : STRING := "i  Sc.Value  Output  Lastvalue  Sc.Time";
  BEGIN
    WRITE (outline1, empty);
    WRITE (outline1, NOW);
    outline2 := outline1;
    WRITELINE (LogFile, outline1);
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (output, outline2);
    END IF;
    WRITE (outline1, header1);
    outline2 := outline1;
    WRITELINE (LogFile, outline1);
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (output, outline2);
    END IF;
    FOR i IN ScheduleDataArray'RANGE LOOP
      WRITE (outline1, i );
      WRITE (outline1, empty);
      DelayValue(ScheduleDataArray(i).InputAge, outline1);
      WRITE (outline1, empty);
      DelayValue(ScheduleDataArray(i).PropDelay, outline1);
      WRITE (outline1, empty);
      DelayValue(ScheduleDataArray(i).OutputRetainDelay, outline1);
      outline2 := outline1;
      WRITELINE (LogFile, outline1);
      IF (PrintDebugMsg = '1') THEN
        WRITELINE (output, outline2);
      END IF;
    END LOOP;
    WRITE (outline1, header2);
    outline2 := outline1;
    WRITELINE (LogFile, outline1);
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (output, outline2);
```

Copyright © 2001 IEEE. All rights reserved.

```
    END IF;
    FOR i IN ScheduleDataArray'RANGE LOOP
      WRITE (outline1, i );
      WRITE (outline1, empty);
      WRITE (outline1, ScheduleDataArray(i).ScheduleValue);
      WRITE (outline1, empty);
      WRITE (outline1, ScheduleDataArray(i).OutputData);
      WRITE (outline1, empty);
      WRITE (outline1, ScheduleDataArray(i).LastOutputValue );
      WRITE (outline1, empty);
      DelayValue(ScheduleDataArray(i).ScheduleTime, outline1);
      outline2 := outline1;
      WRITELINE (LogFile, outline1);
      IF (PrintDebugMsg = '1') THEN
        WRITELINE (output, outline2);
      END IF;
    END LOOP;
    WRITE (outline1, empty);
    WRITE (outline2, empty);
    WRITELINE (LogFile, outline1);
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (Output, outline2);
    END IF;
  END PrintScheduleDataArray;

  -- ----------------------------------------------------------------------------
  PROCEDURE PrintArcType (
    ArcType : IN VitalMemoryArcType
  ) IS
    VARIABLE outline1, outline2 : LINE;
    CONSTANT empty : STRING := " ";
    CONSTANT cross : STRING := "CrossArc";
    CONSTANT para : STRING := "ParallelArc";
    CONSTANT sub : STRING := "SubWordArc";
    CONSTANT Header1 : STRING := "Path considered @ ";
    CONSTANT Header2 : STRING := " is  ";
  BEGIN
    WRITELINE (LogFile, outline1);
    WRITE (outline1, header1);
    WRITE (outline1, NOW);
    WRITE (outline1, empty);
    WRITE (outline1, header2);
    WRITE (outline1, empty);
    case ArcType is
    WHEN CrossArc =>
      WRITE (outline1, cross);
    WHEN ParallelArc =>
      WRITE (outline1, para);
    WHEN SubwordArc =>
      WRITE (outline1, sub);
    END CASE;
    outline2 := outline1 ;
    -- Appears on STD OUT
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (Output, outline1);
    END IF;
    WRITELINE (LogFile, outline2);
  END PrintArcType;

  -- ----------------------------------------------------------------------------
  -- This returns the value picked from the delay array
  -- ----------------------------------------------------------------------------
  PROCEDURE PrintDelay (
    outbitpos         : IN INTEGER;
    InputArrayLow     : IN INTEGER;
    InputArrayHigh    : IN INTEGER;
    debugprop         : IN VitalTimeArrayT;
    debugretain       : IN VitalTimeArrayT
  ) IS
    VARIABLE outline1 : LINE;
    VARIABLE outline2 : LINE;
    VARIABLE outline3 : LINE;
    VARIABLE outline4 : LINE;
    VARIABLE outline5 : LINE;
    VARIABLE outline6 : LINE;
    CONSTANT empty    : STRING := "  ";
    CONSTANT empty5   : STRING := "  ";
    CONSTANT header1  : STRING := "Prop. delays : ";
    CONSTANT header2  : STRING := "Retain delays : ";
    CONSTANT header3  : STRING := "output bit :  ";
  BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
      WRITE(outline1, header3);
      WRITE(outline1, outbitpos);
      outline2 := outline1;
      WRITELINE(LogFile, outline1);
      IF (PrintDebugMsg = '1') THEN
        WRITELINE(output, outline2);
      END IF;
      WRITE(outline1, header1);
      WRITE (outline1, empty5);
      FOR i IN InputArrayHigh DOWNTO InputArrayLow LOOP
        DelayValue(debugprop(i), outline1);
        WRITE(outline1, empty);
      END LOOP;
      outline2 := outline1;
      WRITELINE(LogFile, outline1);
      IF (PrintDebugMsg = '1') THEN
        WRITELINE(output, outline2);
      END IF;
      WRITE(outline1, header2);
      WRITE (outline1, empty5);
      FOR i in InputArrayHigh DOWNTO InputArrayLow LOOP
        DelayValue(debugretain(i), outline1);
        WRITE(outline1, empty);
      END LOOP;
      outline2 := outline1;
      WRITELINE(LogFile, outline1);
      IF (PrintDebugMsg = '1') THEN
        WRITELINE(output, outline2);
      END IF;
    END PrintDelay;

    -- ---------------------------------------------------------------------------
    PROCEDURE DebugMsg1 IS
        CONSTANT header1:STRING:= "******************************************";
        CONSTANT header2 :STRING:="Entering the process because of an i/p change";
        variable outline1, outline2 : LINE;
    BEGIN
        WRITE(outline1, header1);
        outline2 := outline1;
        WRITELINE (Logfile, outline1);
        IF (PrintDebugMsg = '1') THEN
           WRITELINE (output, outline2);
        END IF;
        WRITE(outline1, header2);
        outline2 := outline1;
        WRITELINE (Logfile, outline1);
        IF (PrintDebugMsg = '1') THEN
           WRITELINE (output, outline2);
        END IF;
        WRITE(outline1, header1);
        outline2 := outline1;
        WRITELINE (Logfile, outline1);
        IF (PrintDebugMsg = '1') THEN
           WRITELINE (output, outline2);
        END IF;
    END DebugMsg1;

    -- ---------------------------------------------------------------------------
    PROCEDURE ScheduleDebugMsg IS
      CONSTANT header1  : STRING := "******************************************";
      CONSTANT header2  : STRING := "Finished executing all the procedures";
      VARIABLE outline1 : LINE;
      VARIABLE outline2 : LINE;
    BEGIN
      WRITE(outline1, header1);
      outline2 := outline1;
      IF (PrintDebugMsg = '1') THEN
        WRITELINE (output, outline2);
      END IF;
      WRITELINE (Logfile, outline1);
      WRITE(outline1, header2);
      outline2 := outline1;
      IF (PrintDebugMsg = '1') THEN
        WRITELINE (output, outline2);
      END IF;
      WRITELINE (Logfile, outline1);
      WRITE(outline1, header1);
      outline2 := outline1;
      IF (PrintDebugMsg = '1') THEN
        WRITELINE (output, outline2);
      END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
    WRITELINE (Logfile, outline1);
  END ScheduleDebugMsg;

  -- ---------------------------------------------------------------------------
  PROCEDURE PrintInputName(
    InputSignalName : IN STRING
  ) IS
    VARIABLE outline1 : LINE;
    VARIABLE outline2 : LINE;
    CONSTANT header1  : STRING := "***Changing input is ";
    CONSTANT header2  : STRING := "(";
    CONSTANT header3  : STRING := ")";
    CONSTANT header4  : STRING :=  "****";
    CONSTANT header5  : STRING := "******************************************";
    CONSTANT header6  : STRING:="Entering the process because of an i/p change";
    CONSTANT empty    : STRING := "  ";
  BEGIN
    WRITE(outline1, header5);
    outline2 := outline1;
    WRITELINE (output, outline1);
    WRITELINE (Logfile, outline2);
    WRITE(outline1, header6);
    outline2 := outline1;
    WRITELINE (output, outline1);
    WRITELINE (Logfile, outline2);
    WRITE(outline1, header5);
    outline2 := outline1;
    WRITELINE (output, outline1);
    WRITELINE (Logfile, outline2);
    WRITE(outline1, header1);
    WRITE(outline1, InputSignalName);
    WRITE(outline1, empty);
    WRITE(outline1, now);
    WRITE(outline1, empty);
    WRITE(outline1, header4);
    WRITELINE (output, outline1);
    WRITELINE (Logfile, outline2);
  END PrintInputName;

  -- ---------------------------------------------------------------------------
  PROCEDURE PrintInputChangeTime(
    ChangeTimeArray : IN VitalTimeArrayT
  ) IS
    VARIABLE outline1 : LINE;
    VARIABLE outline2 : LINE;
    CONSTANT header5  : STRING := "*************************************";
    CONSTANT header6  : STRING:="ChangeTime Array : ";
    CONSTANT empty    : STRING := "  ";
  BEGIN
    WRITE(outline1, header5);
    outline2 := outline1;
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (output, outline2);
    END IF;
    WRITELINE (Logfile, outline1);
    WRITE(outline1, header6);
    FOR i in ChangeTimeArray'range LOOP
      WRITE(outline1, ChangeTimeArray(i));
      WRITE(outline1, empty);
    END LOOP;
    outline2 := outline1;
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (output, outline2);
    END IF;
    WRITELINE (Logfile, outline1);
    WRITE(outline1, header5);
    outline2 := outline1;
    IF (PrintDebugMsg = '1') THEN
      WRITELINE (output, outline2);
    END IF;
    WRITELINE (Logfile, outline1);
  END PrintInputChangeTime;

  -- ---------------------------------------------------------------------------
  PROCEDURE PrintInputChangeTime(
    ChangeTime : IN Time
  ) IS
    VARIABLE ChangeTimeArray : VitalTimeArrayT(0 DOWNTO 0);
  BEGIN
    ChangeTimeArray(0) := ChangeTime;
    PrintInputChangeTime(ChangeTimeArray);
```

 Copyright © 2001 IEEE. All rights reserved.

```
END PrintInputChangeTime;

-- ---------------------------------------------------------------------------
-- for debug purpose
CONSTANT MaxNoInputBits : INTEGER := 1000;

TYPE VitalMemoryDelayType IS RECORD
  PropDelay         : TIME;
  OutputRetainDelay : TIME;
END RECORD;

-- ---------------------------------------------------------------------------
-- PROCEDURE:   IntToStr
--
-- PARAMETERS:  InputInt     - Integer to be converted to String.
--              ResultStr    - String buffer for converted Integer
--              AppendPos    - Position in buffer to place result
--
-- DESCRIPTION: This procedure is used to convert an input integer
--              into a string representation. The converted string
--              may be placed at a specific position in the result
--              buffer.
--
-- ---------------------------------------------------------------------------

PROCEDURE IntToStr (
  InputInt    : IN INTEGER ;
  ResultStr   : INOUT STRING ( 1 TO 256) ;
  AppendPos   : INOUT NATURAL
) IS
  -- Look-up table.  Given an int, we can get the character.
  TYPE     integer_table_type IS  ARRAY (0 TO 9) OF CHARACTER ;
  CONSTANT integer_table : integer_table_type :=
    ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9') ;
  --  Local variables used in this function.
  VARIABLE   inpVal      : INTEGER := inputInt ;
  VARIABLE   divisor     : INTEGER := 10 ;
  VARIABLE   tmpStrIndex : INTEGER := 1 ;
  VARIABLE   tmpStr      : STRING ( 1 TO 256 ) ;
BEGIN
  IF ( inpVal = 0 ) THEN
    tmpStr(tmpStrIndex) := integer_table ( 0 ) ;
    tmpStrIndex := tmpStrIndex + 1 ;
  ELSE
    WHILE ( inpVal > 0 )  LOOP
      tmpStr(tmpStrIndex) := integer_table (inpVal mod divisor);
      tmpStrIndex := tmpStrIndex + 1 ;
      inpVal := inpVal / divisor ;
    END LOOP ;
  END IF ;
  IF (appendPos /= 1 ) THEN
    resultStr(appendPos)  :=  ',' ;
    appendPos := appendPos + 1 ;
  END IF ;

  FOR i IN tmpStrIndex-1 DOWNTO 1  LOOP
    resultStr(appendPos)  :=  tmpStr(i) ;
    appendPos := appendPos + 1 ;
  END LOOP ;
END IntToStr ;

-- ---------------------------------------------------------------------------
TYPE CheckType IS (
  SetupCheck,
  HoldCheck,
  RecoveryCheck,
  RemovalCheck,
  PulseWidCheck,
  PeriodCheck
);

TYPE CheckInfoType IS RECORD
  Violation : BOOLEAN;
  CheckKind : CheckType;
  ObsTime   : TIME;
  ExpTime   : TIME;
  DetTime   : TIME;
  State     : X01;
END RECORD;

TYPE LogicCvtTableType IS ARRAY (std_ulogic) OF CHARACTER;
```

Copyright © 2001 IEEE. All rights reserved.

```
TYPE HiLoStrType IS ARRAY (std_ulogic RANGE ‘X’ TO ‘1’) OF STRING(1 TO 4);

CONSTANT LogicCvtTable : LogicCvtTableType
               := ( ‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’);
CONSTANT HiLoStr     : HiLoStrType := (“  X “, “ Low”, “High” );

TYPE EdgeSymbolMatchType IS ARRAY (X01,X01,VitalEdgeSymbolType) OF BOOLEAN;

-- last value, present value, edge symbol
CONSTANT EdgeSymbolMatch : EdgeSymbolMatchType :=
  (
  ‘X’  =>
    ( ‘X’ => (                                 OTHERS => FALSE),
      ‘0’ => (‘N’ |’ F’ |’ v’ |’ E’ |’ D’ |’*’  => TRUE, OTHERS => FALSE ),
      ‘1’ => (‘P’ |’ R’ |’ ^’ |’ E’ |’ A’ |’*’  => TRUE, OTHERS => FALSE )
    ),
  ‘0’  =>
    ( ‘X’ => (      ‘r’ |’ p’ |’ R’ |’ A’ |’*’  => TRUE, OTHERS => FALSE ),
      ‘0’ => (                                 OTHERS => FALSE ),
      ‘1’ => (      ‘/’ |’ P’ |’ p’ |’ R’ |’*’  => TRUE, OTHERS => FALSE )
    ),
  ‘1’  =>
    ( ‘X’ => (      ‘f’ |’ n’ |’ F’ |’ D’ |’*’  => TRUE, OTHERS => FALSE ),
      ‘0’ => (      ‘\’ |’ N’ |’ n’ |’ F’ |’*’  => TRUE, OTHERS => FALSE ),
      ‘1’ => (                                 OTHERS => FALSE )
    )
  );

-- --------------------------------------------------------------------------------
FUNCTION Minimum (
  CONSTANT t1, t2 : IN TIME
) RETURN TIME IS
BEGIN
  IF (t1 < t2) THEN RETURN (t1); ELSE RETURN (t2); END IF;
END Minimum;

-- --------------------------------------------------------------------------------
FUNCTION Maximum (
  CONSTANT t1, t2 : IN TIME
) RETURN TIME IS
BEGIN
  IF (t1 < t2) THEN RETURN (t2); ELSE RETURN (t1); END IF;
END Maximum;

-- --------------------------------------------------------------------------------
-- FUNCTION:    VitalMemoryCalcDelay
-- Description: Select Transition dependent Delay.
--              Used internally by VitalMemorySelectDelay.
-- --------------------------------------------------------------------------------
FUNCTION VitalMemoryCalcDelay (
  CONSTANT NewVal : IN STD_ULOGIC := ‘X’;
  CONSTANT OldVal : IN STD_ULOGIC := ‘X’;
  CONSTANT Delay  : IN VitalDelayType01ZX
) RETURN VitalMemoryDelayType IS
  VARIABLE Result : VitalMemoryDelayType;
BEGIN
  CASE Oldval IS
  WHEN ‘0’ | ‘L’ =>
    CASE Newval IS
      WHEN ‘0’ | ‘L’ =>
        Result.PropDelay := Delay(tr10);
      WHEN ‘1’ | ‘H’ =>
        Result.PropDelay := Delay(tr01);
      WHEN ‘Z’ =>
        Result.PropDelay := Delay(tr0Z);
      WHEN OTHERS =>
        Result.PropDelay := Minimum(Delay(tr01), Delay(tr0Z));
    END CASE;
    Result.OutputRetainDelay := Delay(tr0X);
  WHEN ‘1’ | ‘H’ =>
    CASE Newval IS
      WHEN ‘0’ | ‘L’ =>
        Result.PropDelay := Delay(tr10);
      WHEN ‘1’ | ‘H’ =>
        Result.PropDelay := Delay(tr01);
      WHEN ‘Z’ =>
        Result.PropDelay := Delay(tr1Z);
      WHEN OTHERS =>
        Result.PropDelay := Minimum(Delay(tr10), Delay(tr1Z));
    END CASE;
    Result.OutputRetainDelay := Delay(tr1X);
```

 Copyright © 2001 IEEE. All rights reserved.

```
  WHEN 'Z' =>
    CASE Newval IS
      WHEN '0' | 'L' =>
        Result.PropDelay := Delay(trZ0);
      WHEN '1' | 'H' =>
        Result.PropDelay := Delay(trZ1);
      WHEN 'Z' =>
        Result.PropDelay := Maximum(Delay(tr1Z), Delay(tr0Z));
      WHEN OTHERS =>
        Result.PropDelay := Minimum(Delay(trZ1), Delay(trZ0));
      END CASE;
      Result.OutputRetainDelay := Delay(trZX);
  WHEN OTHERS =>
    CASE Newval IS
      WHEN '0' | 'L' =>
        Result.PropDelay := Maximum(Delay(tr10), Delay(trZ0));
      WHEN '1' | 'H' =>
        Result.PropDelay := Maximum(Delay(tr01), Delay(trZ1));
      WHEN 'Z' =>
        Result.PropDelay := Maximum(Delay(tr1Z), Delay(tr0Z));
      WHEN OTHERS =>
        Result.PropDelay := Maximum(Delay(tr10), Delay(tr01));
    END CASE;
    Result.OutputRetainDelay := Minimum(Delay(tr1X), Delay(tr0X));
  END CASE;
  RETURN Result;
END VitalMemoryCalcDelay;

-- ------------------------------------------------------------------------------
FUNCTION VitalMemoryCalcDelay (
  CONSTANT NewVal : IN STD_ULOGIC   := 'X';
  CONSTANT OldVal : IN STD_ULOGIC   := 'X';
  CONSTANT Delay  : IN VitalDelayType01Z
) RETURN VitalMemoryDelayType IS
  VARIABLE Result : VitalMemoryDelayType;
BEGIN
CASE Oldval IS
  WHEN '0' | 'L' =>
    CASE Newval IS
      WHEN '0' | 'L' => Result.PropDelay := Delay(tr10);
      WHEN '1' | 'H' => Result.PropDelay := Delay(tr01);
      WHEN OTHERS =>
        Result.PropDelay := Minimum(Delay(tr01), Delay(tr10));
    END CASE;
    Result.OutputRetainDelay := Delay(tr0Z);
  WHEN '1' | 'H' =>
    CASE Newval IS
      WHEN '0' | 'L' => Result.PropDelay := Delay(tr10);
      WHEN '1' | 'H' => Result.PropDelay := Delay(tr01);
      WHEN OTHERS =>
        Result.PropDelay := Minimum(Delay(tr10), Delay(tr01));
    END CASE;
    Result.OutputRetainDelay := Delay(tr1Z);
  WHEN OTHERS =>
    Result.PropDelay := Maximum(Delay(tr10),Delay(tr01));
    Result.OutputRetainDelay := Minimum(Delay(tr1Z),Delay(tr0Z));
  END CASE;
  RETURN Result;
END VitalMemoryCalcDelay;

-- ------------------------------------------------------------------------------
PROCEDURE VitalMemoryUpdateInputChangeTime (
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  VARIABLE NumBitsPerSubword    : INTEGER
) IS
  VARIABLE LastInputValue : STD_LOGIC_VECTOR(InputSignal'LENGTH-1 downto 0);
  VARIABLE InSignalNorm   : STD_LOGIC_VECTOR(InputSignal'LENGTH-1 downto 0);
  VARIABLE ChangeTimeNorm : VitalTimeArrayT(InputSignal'LENGTH-1 downto 0);
  VARIABLE BitsPerWord    : INTEGER;
BEGIN
  LastInputValue := InputSignal'LAST_VALUE;
  IF NumBitsPerSubword = DefaultNumBitsPerSubword THEN
    BitsPerWord := InputSignal'LENGTH;
  ELSE
    BitsPerWord := NumBitsPerSubword;
  END IF;

  FOR i IN InSignalNorm'RANGE LOOP
    IF (InSignalNorm(i) /= LastInputValue(i)) THEN
      ChangeTimeNorm(i/BitsPerWord) := NOW - InputSignal'LAST_EVENT;
```

```
      ELSE
        ChangeTimeNorm(i/BitsPerWord) := InputChangeTimeArray(i);
      END IF;
    END LOOP;

    FOR i IN ChangeTimeNorm'RANGE LOOP
      ChangeTimeNorm(i) := ChangeTimeNorm(i/BitsPerword);
    END LOOP;

    InputChangeTimeArray := ChangeTimeNorm;

    -- for debug purpose only
    PrintInputChangeTime(InputChangeTimeArray);
  END VitalMemoryUpdateInputChangeTime;

  -- ----------------------------------------------------------------------------
  -- Procedure:   VitalMemoryUpdateInputChangeTime
  -- Description: Time since previous event for each bit of the input
  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemoryUpdateInputChangeTime (
    VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
    SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR
  ) IS
    VARIABLE LastInputValue  : STD_LOGIC_VECTOR(InputSignal'RANGE) ;
  BEGIN
    LastInputValue := InputSignal'LAST_VALUE;
    FOR i IN InputSignal'RANGE LOOP
      IF (InputSignal(i) /= LastInputValue(i)) THEN
        InputChangeTimeArray(i) := NOW - InputSignal'LAST_EVENT;
      END IF;
    END LOOP;
    -- for debug purpose only
    PrintInputChangeTime(InputChangeTimeArray);
  END VitalMemoryUpdateInputChangeTime;

  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemoryUpdateInputChangeTime (
    VARIABLE InputChangeTime      : INOUT TIME;
    SIGNAL   InputSignal          : IN STD_ULOGIC
  ) IS
  BEGIN
    InputChangeTime := NOW - InputSignal'LAST_EVENT;
    -- for debug purpose only
    PrintInputChangeTime(InputChangeTime);
  END VitalMemoryUpdateInputChangeTime;

  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemoryExpandPortFlag (
    CONSTANT PortFlag             : IN  VitalPortFlagVectorType;
    CONSTANT NumBitsPerSubword    : IN  INTEGER;
    VARIABLE ExpandedPortFlag      : OUT VitalPortFlagVectorType
  ) IS
    VARIABLE PortFlagNorm : VitalPortFlagVectorType(
             PortFlag'LENGTH-1 downto 0) := PortFlag;
    VARIABLE ExpandedPortFlagNorm : VitalPortFlagVectorType(
               ExpandedPortFlag'LENGTH-1 downto 0);
    VARIABLE SubwordIndex : INTEGER;
  BEGIN
    FOR Index IN 0 to ExpandedPortFlag'LENGTH-1 LOOP
      IF NumBitsPerSubword = DefaultNumBitsPerSubword THEN
        SubwordIndex := 0;
      ELSE
        SubwordIndex := Index / NumBitsPerSubword;
      END IF;
      ExpandedPortFlagNorm(Index) := PortFlagNorm(SubWordIndex);
    END LOOP;
    ExpandedPortFlag := ExpandedPortFlagNorm;
  END VitalMemoryExpandPortFlag;

  -- ----------------------------------------------------------------------------
  -- Procedure: VitalMemorySelectDelay
  -- Description : Select Propagation Delay. Used internally by
  --               VitalMemoryAddPathDelay.
  -- ----------------------------------------------------------------------------

  -- ----------------------------------------------------------------------------
  -- VitalDelayArrayType01ZX
  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemorySelectDelay (
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
    VARIABLE InputChangeTimeArray : IN VitalTimeArrayT;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT OutputSignalName     : IN STRING :="";
    CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
    CONSTANT ArcType              : IN VitalMemoryArcType;
    CONSTANT PathConditionArray   : IN VitalBoolArrayT;
    CONSTANT OutputRetainFlag     : IN BOOLEAN
  ) IS
    VARIABLE InputArrayLow      : INTEGER := 0;
    VARIABLE InputArrayHigh     : INTEGER := 0;
    VARIABLE DelayArrayIndex    : INTEGER := 0;
    VARIABLE NumBitsPerSubWord  : INTEGER := DefaultNumBitsPerSubword;
    VARIABLE NewValue           : STD_ULOGIC;
    VARIABLE OldValue           : STD_ULOGIC;
    VARIABLE OutputLength       : INTEGER := 0;
    VARIABLE OutArrayIndex      : INTEGER;
    VARIABLE PropDelay          : TIME;
    VARIABLE RetainDelay        : TIME;
    VARIABLE CurPropDelay       : TIME;
    VARIABLE CurRetainDelay     : TIME;
    VARIABLE InputAge           : TIME;
    VARIABLE CurInputAge        : TIME;
    VARIABLE InputChangeTimeNorm  : VitalTimeArrayT(
       InputChangeTimeArray'LENGTH-1 downto 0):=InputChangeTimeArray;
    VARIABLE DelayArrayNorm  : VitalDelayArrayType01ZX(
       PathDelayArray'LENGTH-1 downto 0):= PathDelayArray;
    VARIABLE ScheduleDataArrayNorm : VitalMemoryScheduleDatavectorType
       (ScheduleDataArray'LENGTH-1 downto 0):= ScheduleDataArray;

    -- for debug purpose
    VARIABLE debugprop : VitalTimeArrayT(MaxNoInputBits-1 downto 0);
    VARIABLE debugretain : VitalTimeArrayT(MaxNoInputBits-1 downto 0);

  BEGIN

    -- for debug purpose
    PrintArcType(ArcType);

    OutputLength := ScheduleDataArray'LENGTH;
    FOR OutBitPos IN 0 to (OutputLength -1) LOOP
      NEXT WHEN PathConditionArray(OutBitPos) = FALSE;

      NEXT WHEN ((ScheduleDataArrayNorm(OutBitPos).ScheduleValue
        = ScheduleDataArrayNorm(OutBitPos).OutputData) AND
        (ScheduleDataArrayNorm(OutBitPos).ScheduleTime <= NOW) AND
         (OutputRetainFlag = FALSE ));

      NewValue := ScheduleDataArrayNorm(OutBitPos).OutputData;
      OldValue := ScheduleDataArrayNorm(OutBitPos).LastOutputValue;
      PropDelay :=ScheduleDataArrayNorm(OutBitPos).PropDelay;
      InputAge := ScheduleDataArrayNorm(OutBitPos).InputAge;
      RetainDelay:=ScheduleDataArrayNorm(OutBitPos).OutputRetainDelay;
      NumBitsPerSubWord:=ScheduleDataArrayNorm(OutBitPos).NumBitsPerSubWord;

      CASE ArcType IS
        WHEN ParallelArc =>
          InputArrayLow := OutBitPos;
          InputArrayHigh :=  OutBitPos;
          DelayArrayIndex :=  OutBitPos;
        WHEN CrossArc =>
          InputArrayLow := 0;
          InputArrayHigh := InputChangeTimeArray'LENGTH - 1 ;
          DelayArrayIndex := OutBitPos;
        WHEN SubwordArc =>
          InputArrayLow := OutBitPos / NumBitsPerSubWord;
          InputArrayHigh := OutBitPos / NumBitsPerSubWord;
          DelayArrayIndex := OutBitPos +
              (OutputLength *  (OutBitPos / NumBitsPerSubWord));
      END CASE;

      FOR i IN InputArrayLow TO InputArrayHigh LOOP
        (CurPropDelay,CurRetainDelay) :=
          VitalMemoryCalcDelay (
            NewValue, OldValue, DelayArrayNorm(DelayArrayIndex)
          );
        IF (OutputRetainFlag = FALSE) THEN
          CurRetainDelay := TIME'HIGH;
        END IF;

        -- for debug purpose
        debugprop(i) := CurPropDelay;
        debugretain(i) := CurRetainDelay;
```

Copyright © 2001 IEEE. All rights reserved.

```
      IF ArcType = CrossArc THEN
        DelayArrayIndex := DelayArrayIndex + OutputLength;
      END IF;

      -- If there is one input change at a time, then choose the
      -- delay from that input. If there is simultaneous input
      -- change, then choose the minimum of propagation delays

      IF (InputChangeTimeNorm(i) < 0 ns)THEN
        CurInputAge := TIME'HIGH;
      ELSE
        CurInputAge := NOW - InputChangeTimeNorm(i);
      END IF;

      IF (CurInputAge  < InputAge)THEN
        PropDelay := CurPropDelay;
        RetainDelay := CurRetainDelay;
        InputAge := CurInputAge;
      ELSIF (CurInputAge = InputAge)THEN
        IF (CurPropDelay < PropDelay) THEN
          PropDelay := CurPropDelay;
        END IF;
        IF (OutputRetainFlag = TRUE) THEN
          IF (CurRetainDelay < RetainDelay) THEN
            RetainDelay := CurRetainDelay;
          END IF;
        END IF;
      END IF;
    END LOOP;

    -- Store it back to data strucutre
    ScheduleDataArrayNorm(OutBitPos).PropDelay := PropDelay;
    ScheduleDataArrayNorm(OutBitPos).OutputRetainDelay:= RetainDelay;
    ScheduleDataArrayNorm(OutBitPos).InputAge := InputAge;

    -- for debug purpose
    PrintDelay(outbitPos,InputArrayLow, InputArrayHigh,
      debugprop, debugretain);
  END LOOP;

  ScheduleDataArray := ScheduleDataArrayNorm;

END VitalMemorySelectDelay;

-- ----------------------------------------------------------------------------
-- VitalDelayArrayType01Z
-- ----------------------------------------------------------------------------
PROCEDURE VitalMemorySelectDelay (
  VARIABLE ScheduleDataArray    : INOUT VitalMemoryScheduleDataVectorType;
  VARIABLE InputChangeTimeArray : IN VitalTimeArrayT;
  CONSTANT OutputSignalName     : IN STRING :="";
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag     : IN BOOLEAN
) IS
  VARIABLE InputArrayLow      : INTEGER := 0;
  VARIABLE InputArrayHigh     : INTEGER := 0;
  VARIABLE DelayArrayIndex    : INTEGER := 0;
  VARIABLE NumBitsPerSubWord  : INTEGER := DefaultNumBitsPerSubword;
  VARIABLE NewValue           : STD_ULOGIC;
  VARIABLE OldValue           : STD_ULOGIC;
  VARIABLE OutputLength       : INTEGER := 0;
  VARIABLE OutArrayIndex      : INTEGER;
  VARIABLE PropDelay          : TIME;
  VARIABLE RetainDelay        : TIME;
  VARIABLE CurPropDelay       : TIME;
  VARIABLE CurRetainDelay     : TIME;
  VARIABLE InputAge           : TIME;
  VARIABLE CurInputAge        : TIME;
  VARIABLE InputChangeTimeNorm  : VitalTimeArrayT(
     InputChangeTimeArray'LENGTH-1 downto 0):=InputChangeTimeArray;
  VARIABLE DelayArrayNorm  : VitalDelayArrayType01Z(
     PathDelayArray'LENGTH-1 downto 0):= PathDelayArray;
  VARIABLE ScheduleDataArrayNorm : VitalMemoryScheduleDatavectorType
     (ScheduleDataArray'LENGTH-1 downto 0):=ScheduleDataArray;

  -- for debug purpose
  VARIABLE debugprop : VitalTimeArrayT(MaxNoInputBits-1 downto 0);
  VARIABLE debugretain : VitalTimeArrayT(MaxNoInputBits-1 downto 0);
BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- for debug purpose
PrintArcType(ArcType);

OutputLength := ScheduleDataArray'LENGTH;
FOR OutBitPos IN 0 to (OutputLength -1) LOOP
  NEXT WHEN PathConditionArray(OutBitPos) = FALSE;

  NEXT WHEN ((ScheduleDataArrayNorm(OutBitPos).ScheduleValue
    = ScheduleDataArrayNorm(OutBitPos).OutputData) AND
      (ScheduleDataArrayNorm(OutBitPos).ScheduleTime <= NOW) AND
      (OutputRetainFlag = FALSE));

  NewValue := ScheduleDataArrayNorm(OutBitPos).OutputData;
  OldValue := ScheduleDataArrayNorm(OutBitPos).LastOutputValue;
  PropDelay :=ScheduleDataArrayNorm(OutBitPos).PropDelay;
  InputAge := ScheduleDataArrayNorm(OutBitPos).InputAge;
  RetainDelay:=ScheduleDataArrayNorm(OutBitPos).OutputRetainDelay;
  NumBitsPerSubWord:=ScheduleDataArrayNorm(OutBitPos).NumBitsPerSubWord;

  CASE ArcType IS
    WHEN ParallelArc =>
      InputArrayLow := OutBitPos;
      InputArrayHigh :=  OutBitPos;
      DelayArrayIndex :=  OutBitPos;
    WHEN CrossArc =>
      InputArrayLow := 0;
      InputArrayHigh := InputChangeTimeArray'LENGTH-1;
      DelayArrayIndex := OutBitPos;
    WHEN SubwordArc =>
      InputArrayLow := OutBitPos / NumBitsPerSubWord;
      InputArrayHigh := OutBitPos / NumBitsPerSubWord;
      DelayArrayIndex := OutBitPos +
        (OutputLength *  (OutBitPos / NumBitsPerSubWord));
  END CASE;

  FOR i IN InputArrayLow TO InputArrayHigh LOOP
    (CurPropDelay, CurRetainDelay) :=
      VitalMemoryCalcDelay (
        NewValue, OldValue, DelayArrayNorm(DelayArrayIndex)
      );
    IF (OutputRetainFlag = FALSE) THEN
      CurRetainDelay := TIME'HIGH;
    END IF;

    -- for debug purpose
    debugprop(i) := CurPropDelay;
    debugretain(i) := CurRetainDelay;

    IF (ArcType = CrossArc) THEN
      DelayArrayIndex := DelayArrayIndex + OutputLength;
    END IF;

    -- If there is one input change at a time, then choose the
    -- delay from that input. If there is simultaneous input
    -- change, then choose the minimum of propagation delays

    IF (InputChangeTimeNorm(i) < 0 ns) THEN
      CurInputAge := TIME'HIGH;
    ELSE
      CurInputAge := NOW - InputChangeTimeNorm(i);
    END IF;

    IF (CurInputAge  < InputAge) THEN
      PropDelay := CurPropDelay;
      RetainDelay := CurRetainDelay;
      InputAge := CurInputAge;
    ELSIF (CurInputAge = InputAge) THEN
      IF (CurPropDelay < PropDelay) THEN
        PropDelay := CurPropDelay;
      END IF;
      IF (OutputRetainFlag = TRUE) THEN
        IF (CurRetainDelay < RetainDelay) THEN
          RetainDelay := CurRetainDelay;
        END IF;
      END IF;
    END IF;
  END LOOP;

  -- Store it back to data strucutre
  ScheduleDataArrayNorm(OutBitPos).PropDelay := PropDelay;
```

Copyright © 2001 IEEE. All rights reserved.

```
      ScheduleDataArrayNorm(OutBitPos).OutputRetainDelay:= RetainDelay;
      ScheduleDataArrayNorm(OutBitPos).InputAge := InputAge;

      -- for debug purpose
      PrintDelay(outbitPos, InputArrayLow, InputArrayHigh,
         debugprop, debugretain);
    END LOOP;

    ScheduleDataArray := ScheduleDataArrayNorm;

  END VitalMemorySelectDelay;

  -- ----------------------------------------------------------------------------
  -- VitalDelayArrayType01
  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemorySelectDelay (
    VARIABLE ScheduleDataArray    : INOUT VitalMemoryScheduleDataVectorType;
    VARIABLE InputChangeTimeArray : IN    VitalTimeArrayT;
    CONSTANT OutputSignalName     : IN    STRING :="";
    CONSTANT PathDelayArray       : IN    VitalDelayArrayType01;
    CONSTANT ArcType              : IN    VitalMemoryArcType;
    CONSTANT PathConditionArray   : IN VitalBoolArrayT
  ) IS
    VARIABLE CurPathDelay       : VitalMemoryDelayType;
    VARIABLE InputArrayLow      : INTEGER := 0;
    VARIABLE InputArrayHigh     : INTEGER := 0;
    VARIABLE DelayArrayIndex    : INTEGER := 0;
    VARIABLE NumBitsPerSubWord  : INTEGER := DefaultNumBitsPerSubword;
    VARIABLE NewValue           : STD_ULOGIC;
    VARIABLE OldValue           : STD_ULOGIC;
    VARIABLE OutputLength       : INTEGER := 0;
    VARIABLE OutArrayIndex      : INTEGER;
    VARIABLE PropDelay          : TIME;
    VARIABLE CurPropDelay       : TIME;
    VARIABLE InputAge           : TIME;
    VARIABLE CurInputAge        : TIME;
    VARIABLE InputChangeTimeNorm  : VitalTimeArrayT(
       InputChangeTimeArray'LENGTH-1 downto 0):= InputChangeTimeArray;
    VARIABLE DelayArrayNorm  : VitalDelayArrayType01(
       PathDelayArray'LENGTH-1 downto 0):= PathDelayArray;
    VARIABLE ScheduleDataArrayNorm : VitalMemoryScheduleDatavectorType
       (ScheduleDataArray'LENGTH-1 downto 0):=ScheduleDataArray;

    -- for debug purpose
    VARIABLE debugprop : VitalTimeArrayT(MaxNoInputBits-1 downto 0);
    VARIABLE debugretain : VitalTimeArrayT(MaxNoInputBits-1 downto 0);
  BEGIN

    -- for debug purpose
    PrintArcType(ArcType);

    OutputLength := ScheduleDataArray'LENGTH;
    FOR OutBitPos IN 0 to (OutputLength -1) LOOP
      NEXT WHEN PathConditionArray(OutBitPos) = FALSE;

      NEXT WHEN ((ScheduleDataArrayNorm(OutBitPos).ScheduleValue
      = ScheduleDataArrayNorm(OutBitPos).OutputData) AND
      (ScheduleDataArrayNorm(OutBitPos).ScheduleTime <= NOW));

      NewValue := ScheduleDataArrayNorm(OutBitPos).OutputData;
      OldValue := ScheduleDataArrayNorm(OutBitPos).LastOutputValue;
      PropDelay :=ScheduleDataArrayNorm(OutBitPos).PropDelay;
      InputAge := ScheduleDataArrayNorm(OutBitPos).InputAge;
      NumBitsPerSubWord:=ScheduleDataArrayNorm(OutBitPos).NumBitsPerSubWord;

      CASE ArcType IS
        WHEN ParallelArc =>
          InputArrayLow := OutBitPos;
          InputArrayHigh :=  OutBitPos;
          DelayArrayIndex :=  OutBitPos;
        WHEN CrossArc =>
          InputArrayLow := 0;
          InputArrayHigh := InputChangeTimeArray'LENGTH-1;
          DelayArrayIndex := OutBitPos;
        WHEN SubwordArc =>
          InputArrayLow := OutBitPos / NumBitsPerSubWord;
          InputArrayHigh := OutBitPos / NumBitsPerSubWord;
          DelayArrayIndex := OutBitPos +
            (OutputLength *  (OutBitPos / NumBitsPerSubWord));
      END CASE;
```

 Copyright © 2001 IEEE. All rights reserved.

```
      FOR i IN InputArrayLow TO InputArrayHigh LOOP
        CurPropDelay:= VitalCalcDelay (NewValue,
                    OldValue, DelayArrayNorm(DelayArrayIndex));

        -- for debug purpose
        debugprop(i) := CurPropDelay;
        debugretain(i) := TIME'HIGH;

        IF (ArcType = CrossArc) THEN
          DelayArrayIndex := DelayArrayIndex + OutputLength;
        END IF;

        -- If there is one input change at a time, then choose the
        -- delay from that input. If there is simultaneous input
        -- change, then choose the minimum of propagation delays

        IF (InputChangeTimeNorm(i) < 0 ns) THEN
          CurInputAge := TIME'HIGH;
        ELSE
          CurInputAge := NOW - InputChangeTimeNorm(i);
        END IF;
        IF (CurInputAge  < InputAge) THEN
          PropDelay := CurPropDelay;
          InputAge := CurInputAge;
        ELSIF (CurInputAge = InputAge) THEN
          IF (CurPropDelay < PropDelay) THEN
            PropDelay := CurPropDelay;
          END IF;
        END IF;
      END LOOP;

      -- Store it back to data strucutre
      ScheduleDataArrayNorm(OutBitPos).PropDelay := PropDelay;
      ScheduleDataArrayNorm(OutBitPos).InputAge := InputAge;

      -- for debug purpose
      PrintDelay(outbitPos, InputArrayLow, InputArrayHigh,
        debugprop, debugretain);
    END LOOP;

    ScheduleDataArray := ScheduleDataArrayNorm;

  END VitalMemorySelectDelay;

  -- ------------------------------------------------------------------------------
  -- VitalDelayArrayType
  -- ------------------------------------------------------------------------------
  PROCEDURE VitalMemorySelectDelay (
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
    VARIABLE InputChangeTimeArray : IN VitalTimeArrayT;
    CONSTANT OutputSignalName     : IN STRING :="";
    CONSTANT PathDelayArray       : IN VitalDelayArrayType;
    CONSTANT ArcType              : IN VitalMemoryArcType;
    CONSTANT PathConditionArray   : IN VitalBoolArrayT
  ) IS
    VARIABLE InputArrayLow      : INTEGER := 0;
    VARIABLE InputArrayHigh     : INTEGER := 0;
    VARIABLE DelayArrayIndex    : INTEGER := 0;
    VARIABLE NumBitsPerSubWord  : INTEGER := DefaultNumBitsPerSubword;
    VARIABLE NewValue           : STD_ULOGIC;
    VARIABLE OldValue           : STD_ULOGIC;
    VARIABLE OutputLength       : INTEGER := 0;
    VARIABLE OutArrayIndex      : INTEGER;
    VARIABLE PropDelay          : TIME;
    VARIABLE CurPropDelay       : TIME;
    VARIABLE InputAge           : TIME;
    VARIABLE CurInputAge        : TIME;
    VARIABLE InputChangeTimeNorm  : VitalTimeArrayT(
       InputChangeTimeArray'LENGTH-1 downto 0) := InputChangeTimeArray;
    VARIABLE DelayArrayNorm  : VitalDelayArrayType(
       PathDelayArray'LENGTH-1 downto 0) := PathDelayArray;
    VARIABLE ScheduleDataArrayNorm : VitalMemoryScheduleDatavectorType
       (ScheduleDataArray'LENGTH-1 downto 0) := ScheduleDataArray;

    -- for debug purpose
    VARIABLE debugprop : VitalTimeArrayT(MaxNoInputBits-1 downto 0);
    VARIABLE debugretain : VitalTimeArrayT(MaxNoInputBits-1 downto 0);
  BEGIN

    -- for debug purpose
    PrintArcType(ArcType);
```


Copyright © 2001 IEEE. All rights reserved.

```
  OutputLength := ScheduleDataArray'LENGTH;
  FOR OutBitPos IN 0 to (OutputLength -1) LOOP
    NEXT WHEN PathConditionArray(OutBitPos) = FALSE;

    NEXT WHEN ((ScheduleDataArrayNorm(OutBitPos).ScheduleValue
      = ScheduleDataArrayNorm(OutBitPos).OutputData) AND
        (ScheduleDataArrayNorm(OutBitPos).ScheduleTime <= NOW));

    NewValue := ScheduleDataArrayNorm(OutBitPos).OutputData;
    OldValue := ScheduleDataArrayNorm(OutBitPos).LastOutputValue;
    PropDelay :=ScheduleDataArrayNorm(OutBitPos).PropDelay;
    InputAge := ScheduleDataArrayNorm(OutBitPos).InputAge;
    NumBitsPerSubWord:=ScheduleDataArrayNorm(OutBitPos).NumBitsPerSubWord;

    CASE ArcType IS
      WHEN ParallelArc =>
        InputArrayLow := OutBitPos;
        InputArrayHigh :=  OutBitPos;
        DelayArrayIndex :=  OutBitPos;
      WHEN CrossArc =>
        InputArrayLow := 0;
        InputArrayHigh := InputChangeTimeArray'LENGTH-1;
        DelayArrayIndex := OutBitPos;
      WHEN SubwordArc =>
        InputArrayLow := OutBitPos / NumBitsPerSubWord;
        InputArrayHigh := OutBitPos / NumBitsPerSubWord;
        DelayArrayIndex := OutBitPos +
          (OutputLength *  (OutBitPos / NumBitsPerSubWord));
    END CASE;

    FOR i IN InputArrayLow TO InputArrayHigh LOOP
      CurPropDelay := VitalCalcDelay (NewValue,
        OldValue, DelayArrayNorm(DelayArrayIndex));

      -- for debug purpose
      debugprop(i) := CurPropDelay;
      debugretain(i) := TIME'HIGH;

      IF (ArcType = CrossArc) THEN
        DelayArrayIndex := DelayArrayIndex + OutputLength;
      END IF;

      -- If there is one input change at a time, then choose the
      -- delay from that input. If there is simultaneous input
      -- change, then choose the minimum of propagation delays

      IF (InputChangeTimeNorm(i) < 0 ns) THEN
        CurInputAge := TIME'HIGH;
      ELSE
        CurInputAge := NOW - InputChangeTimeNorm(i);
      END IF;

      IF (CurInputAge  < InputAge) THEN
        PropDelay := CurPropDelay;
        InputAge := CurInputAge;
      ELSIF (CurInputAge = InputAge) THEN
        IF (CurPropDelay < PropDelay) THEN
          PropDelay := CurPropDelay;
        END IF;
      END IF;
    END LOOP;

    -- Store it back to data strucutre
    ScheduleDataArrayNorm(OutBitPos).PropDelay := PropDelay;
    ScheduleDataArrayNorm(OutBitPos).InputAge := InputAge;

    -- for debug purpose
    PrintDelay(outbitPos, InputArrayLow, InputArrayHigh,
      debugprop, debugretain);
  END LOOP;

  ScheduleDataArray := ScheduleDataArrayNorm;

END VitalMemorySelectDelay;

-- ----------------------------------------------------------------------------
-- Procedure:   VitalMemoryInitPathDelay
-- Description: To initialize Schedule Data structure for an
--              output.
-- ----------------------------------------------------------------------------
```

 Copyright © 2001 IEEE. All rights reserved.

```
PROCEDURE VitalMemoryInitPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  VARIABLE OutputDataArray   : IN STD_LOGIC_VECTOR;
  CONSTANT NumBitsPerSubWord : IN INTEGER := DefaultNumBitsPerSubword
) IS
BEGIN
  -- Initialize the ScheduleData Structure.
  FOR i IN OutputDataArray'RANGE LOOP
    ScheduleDataArray(i).OutputData    := OutputDataArray(i);
    ScheduleDataArray(i).PropDelay     := TIME'HIGH;
    ScheduleDataArray(i).OutputRetainDelay := TIME'HIGH;
    ScheduleDataArray(i).InputAge      := TIME'HIGH;
    ScheduleDataArray(i).NumBitsPerSubWord := NumBitsPerSubWord;

    -- Update LastOutputValue of Output if the Output has
    -- already been scheduled.
    IF ((ScheduleDataArray(i).ScheduleValue /= OutputDataArray(i)) AND
        (ScheduleDataArray(i).ScheduleTime <= NOW)) THEN
       ScheduleDataArray(i).LastOutputValue
                := ScheduleDataArray(i).ScheduleValue;
    END IF;
  END LOOP;

  -- for debug purpose
  DebugMsg1;
  PrintScheduleDataArray(ScheduleDataArray);

END VitalMemoryInitPathDelay;

-- ----------------------------------------------------------------------------
PROCEDURE VitalMemoryInitPathDelay (
  VARIABLE ScheduleData : INOUT VitalMemoryScheduleDataType;
  VARIABLE OutputData   : IN STD_ULOGIC
) IS
  VARIABLE ScheduledataArray: VitalMemoryScheduleDataVectorType
                                      (0 downto 0);
  VARIABLE OutputDataArray  : STD_LOGIC_VECTOR(0 downto 0);
BEGIN
  ScheduledataArray(0) := ScheduleData;
  OutputDataArray(0)   := OutputData;
  VitalMemoryInitPathDelay (
    ScheduleDataArray => ScheduleDataArray,
    OutputDataArray   => OutputDataArray,
    NumBitsPerSubWord => DefaultNumBitsPerSubword
  );

  -- for debug purpose
  DebugMsg1;
  PrintScheduleDataArray( ScheduleDataArray);

END VitalMemoryInitPathDelay;

-- ----------------------------------------------------------------------------
-- Procedure:   VitalMemoryAddPathDelay
-- Description: Declare a path for one scalar/vector input to
--              the output for which Schedule Data has been
--              initialized previously.
-- ----------------------------------------------------------------------------

-- ----------------------------------------------------------------------------
-- #1
-- DelayType - VitalMemoryDelayType
-- Input     - Scalar
-- Output    - Scalar
-- Delay     - Scalar
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData      : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTime   : INOUT TIME;
  CONSTANT PathDelay         : IN VitalDelayType;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE
) IS
  VARIABLE ScheduleDataArray :
             VitalMemoryScheduleDataVectorType(0 downto 0);
  VARIABLE PathDelayArray : VitalDelayArrayType(0 downto 0);
  VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
  VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
BEGIN
```

Copyright © 2001 IEEE. All rights reserved.

```
      PathConditionArray(0) := PathCondition;
      ScheduleDataArray(0) := ScheduleData;
      PathDelayArray(0) := PathDelay;
      VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
      InputChangeTimeArray(0) := InputChangeTime;

      VitalMemorySelectDelay(
        ScheduleDataArray, InputChangeTimeArray,
        OutputSignalName, PathDelayArray,
        ArcType, PathConditionArray);
    END VitalMemoryAddPathDelay;

    -- ----------------------------------------------------------------------------
    -- #2
    -- DelayType - VitalMemoryDelayType
    -- Input     - Scalar
    -- Output    - Vector
    -- Delay     - Vector
    -- Condition - Scalar
    PROCEDURE VitalMemoryAddPathDelay (
      VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
      SIGNAL   InputSignal        : IN STD_ULOGIC;
      CONSTANT OutputSignalName   : IN STRING :="";
      VARIABLE InputChangeTime    : INOUT TIME;
      CONSTANT PathDelayArray     : IN VitalDelayArrayType;
      CONSTANT ArcType            : IN VitalMemoryArcType := CrossArc;
      CONSTANT PathCondition      : IN BOOLEAN := TRUE
    ) IS
      VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
      VARIABLE PathConditionArray :
          VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
    BEGIN
      FOR i IN PathConditionArray'RANGE LOOP
        PathConditionArray(i) := PathCondition;
      END LOOP;

      VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
      InputChangeTimeArray(0) := InputChangeTime;

      VitalMemorySelectDelay(
        ScheduleDataArray, InputChangeTimeArray,
        OutputSignalName, PathDelayArray,
        ArcType, PathConditionArray
      );
    END VitalMemoryAddPathDelay;

    -- ----------------------------------------------------------------------------
    -- #3
    -- DelayType - VitalMemoryDelayType
    -- Input     - Scalar
    -- Output    - Vector
    -- Delay     - Vector
    -- Condition - Vector
    PROCEDURE VitalMemoryAddPathDelay (
      VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
      SIGNAL   InputSignal        : IN STD_ULOGIC;
      CONSTANT OutputSignalName   : IN STRING :="";
      VARIABLE InputChangeTime    : INOUT TIME;
      CONSTANT PathDelayArray     : IN VitalDelayArrayType;
      CONSTANT ArcType            : IN VitalMemoryArcType := CrossArc;
      CONSTANT PathConditionArray : IN VitalBoolArrayT
    ) IS
      VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
      VARIABLE NumBitsPerSubword : INTEGER;
      VARIABLE PathConditionArrayNorm :
          VitalBoolArrayT(PathConditionArray'LENGTH-1 downto 0);
      VARIABLE PathConditionArrayExp :
          VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
    BEGIN
      NumBitsPerSubword :=
        ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
      FOR i IN PathConditionArrayExp'RANGE LOOP
        PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
      END LOOP;

      VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
      InputChangeTimeArray(0) := InputChangeTime;

      VitalMemorySelectDelay(
        ScheduleDataArray, InputChangeTimeArray,
        OutputSignalName, PathDelayArray,
```

 Copyright © 2001 IEEE. All rights reserved.

```
    ArcType, PathConditionArrayExp);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #4
-- DelayType - VitalMemoryDelayType
-- Input     - Vector
-- Output    - Scalar
-- Delay     - Vector
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING :="";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE
) IS
  VARIABLE ScheduleDataArray : VitalMemoryScheduleDataVectorType(0 downto 0);
  VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
BEGIN
  PathConditionArray(0) := PathCondition;

  ScheduleDataArray(0) := ScheduleData;
  VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #5
-- DelayType - VitalMemoryDelayType
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal        : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName   : IN STRING :="";
  VARIABLE InputChangeTimeArray  : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray        : IN VitalDelayArrayType;
  CONSTANT ArcType               : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition         : IN BOOLEAN := TRUE
) IS
  VARIABLE PathConditionArray :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
BEGIN
  FOR i IN PathConditionArray'RANGE LOOP
    PathConditionArray(i) := PathCondition;
  END LOOP;

  VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #6
-- DelayType - VitalMemoryDelayType
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal        : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName   : IN STRING :="";
  VARIABLE InputChangeTimeArray  : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray        : IN VitalDelayArrayType;
  CONSTANT ArcType               : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray    : IN VitalBoolArrayT
) IS
  VARIABLE NumBitsPerSubword : INTEGER;
```

Copyright © 2001 IEEE. All rights reserved.

```
  VARIABLE PathConditionArrayNorm :
      VitalBoolArrayT(PathConditionArray'LENGTH-1 downto 0);
  VARIABLE PathConditionArrayExp :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
BEGIN
  NumBitsPerSubword :=
      ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
  FOR i IN PathConditionArrayExp'RANGE LOOP
    PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
  END LOOP;

  VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArrayExp);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #7
-- DelayType - VitalMemoryDelayType01
-- Input     - Scalar
-- Output    - Scalar
-- Delay     - Scalar
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData      : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTime   : INOUT TIME;
  CONSTANT PathDelay         : IN VitalDelayType01;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE
) IS
  VARIABLE ScheduleDataArray :
         VitalMemoryScheduleDataVectorType(0 downto 0);
  VARIABLE PathDelayArray : VitalDelayArrayType01(0 downto 0);
  VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
  VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
BEGIN
  PathConditionArray(0) := PathCondition;
  ScheduleDataArray(0) := ScheduleData;
  PathDelayArray(0) := PathDelay;
  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #8
-- DelayType - VitalMemoryDelayType01
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTime   : INOUT TIME;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE
) IS
  VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
  VARIABLE PathConditionArray :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
BEGIN
  FOR i IN PathConditionArray'RANGE LOOP
    PathConditionArray(i) := PathCondition;
  END LOOP;

  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
```

 Copyright © 2001 IEEE. All rights reserved.

```
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArray);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #9
  -- DelayType - VitalMemoryDelayType01
  -- Input     - Scalar
  -- Output    - Vector
  -- Delay     - Vector
  -- Condition - Vector
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
    SIGNAL   InputSignal       : IN STD_ULOGIC;
    CONSTANT OutputSignalName  : IN STRING :="";
    VARIABLE InputChangeTime   : INOUT TIME;
    CONSTANT PathDelayArray    : IN VitalDelayArrayType01;
    CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
    CONSTANT PathConditionArray: IN VitalBoolArrayT
  ) IS
    VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
    VARIABLE NumBitsPerSubword : INTEGER;
    VARIABLE PathConditionArrayNorm :
        VitalBoolArrayT(PathConditionArray'LENGTH-1 downto 0);
    VARIABLE PathConditionArrayExp :
        VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
  BEGIN
    NumBitsPerSubword :=
        ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
    FOR i IN PathConditionArrayExp'RANGE LOOP
      PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
    END LOOP;

    VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
    InputChangeTimeArray(0) := InputChangeTime;

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArrayExp);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #10
  -- DelayType - VitalMemoryDelayType01
  -- Input     - Vector
  -- Output    - Scalar
  -- Delay     - Vector
  -- Condition - Scalar
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleData        : INOUT VitalMemoryScheduleDataType;
    SIGNAL   InputSignal         : IN STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName    : IN STRING :="";
    VARIABLE InputChangeTimeArray: INOUT VitalTimeArrayT;
    CONSTANT PathDelayArray      : IN VitalDelayArrayType01;
    CONSTANT ArcType             : IN VitalMemoryArcType := CrossArc;
    CONSTANT PathCondition       : IN BOOLEAN := TRUE
  )IS
    VARIABLE ScheduleDataArray :
        VitalMemoryScheduleDataVectorType(0 downto 0);
    VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
  BEGIN
    PathConditionArray(0) := PathCondition;
    ScheduleDataArray(0) := ScheduleData;
    VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArray);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #11
  -- DelayType - VitalMemoryDelayType01
  -- Input     - Vector
  -- Output    - Vector
  -- Delay     - Vector
  -- Condition - Scalar
  PROCEDURE VitalMemoryAddPathDelay (
```

Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
    SIGNAL   InputSignal        : IN STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName   : IN STRING :="";
    VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
    CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
    CONSTANT ArcType              : IN VitalMemoryArcType  := CrossArc;
    CONSTANT PathCondition        : IN BOOLEAN := TRUE
  ) IS
    VARIABLE PathConditionArray :
        VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
  BEGIN
    FOR i IN PathConditionArray'RANGE LOOP
      PathConditionArray(i) := PathCondition;
    END LOOP;

    VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArray);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #12
  -- DelayType - VitalMemoryDelayType01
  -- Input     - Vector
  -- Output    - Vector
  -- Delay     - Vector
  -- Condition - Vector
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
    SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName  : IN STRING :="";
    VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
    CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
    CONSTANT ArcType              : IN VitalMemoryArcType  := CrossArc;
    CONSTANT PathConditionArray   : IN VitalBoolArrayT
  ) IS
    VARIABLE NumBitsPerSubword : INTEGER;
    VARIABLE PathConditionArrayNorm :
        VitalBoolArrayT(PathConditionArray'LENGTH-1 downto 0);
    VARIABLE PathConditionArrayExp :
        VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
  BEGIN
    NumBitsPerSubword :=
        ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
    FOR i IN PathConditionArrayExp'RANGE LOOP
      PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
    END LOOP;

    VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArrayExp);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #13
  -- DelayType - VitalMemoryDelayType01Z
  -- Input     - Scalar
  -- Output    - Scalar
  -- Delay     - Scalar
  -- Condition - Scalar
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleData     : INOUT VitalMemoryScheduleDataType;
    SIGNAL   InputSignal      : IN STD_ULOGIC;
    CONSTANT OutputSignalName : IN STRING :="";
    VARIABLE InputChangeTime  : INOUT TIME;
    CONSTANT PathDelay        : IN VitalDelayType01Z;
    CONSTANT ArcType          : IN VitalMemoryArcType := CrossArc;
    CONSTANT PathCondition    : IN BOOLEAN := TRUE;
    CONSTANT OutputRetainFlag : IN BOOLEAN := FALSE
  ) IS
    VARIABLE ScheduleDataArray :
             VitalMemoryScheduleDataVectorType(0 downto 0);
    VARIABLE PathDelayArray : VitalDelayArrayType01Z(0 downto 0);
    VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
    VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
```

 Copyright © 2001 IEEE. All rights reserved.

```
BEGIN
  PathConditionArray(0) := PathCondition;
  ScheduleDataArray(0) := ScheduleData;
  PathDelayArray(0) := PathDelay;
  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray, OutputRetainFlag);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #14
-- DelayType - VitalMemoryDelayType01Z
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTime   : INOUT TIME;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01Z;
  CONSTANT ArcType           : IN VitalMemoryArcType  := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
) IS
  VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
  VARIABLE PathConditionArray :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
BEGIN
  FOR i IN PathConditionArray'RANGE LOOP
    PathConditionArray(i) := PathCondition;
  END LOOP;

  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray, OutputRetainFlag);

END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #15
-- DelayType - VitalMemoryDelayType01Z
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTime   : INOUT TIME;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01Z;
  CONSTANT ArcType           : IN VitalMemoryArcType  := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
) IS
  VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
  VARIABLE NumBitsPerSubword : INTEGER;
  VARIABLE PathConditionArrayNorm : VitalBoolArrayT(PathConditionArray'LENGTH-1 downto
0);
   VARIABLE  PathConditionArrayExp  :  VitalBoolArrayT(ScheduleDataArray'LENGTH-1  downto
0);
BEGIN
  NumBitsPerSubword := ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
  FOR i IN PathConditionArrayExp'RANGE LOOP
    PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
  END LOOP;

  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
```

Copyright © 2001 IEEE. All rights reserved.

```
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArrayExp, OutputRetainFlag);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #16
-- DelayType - VitalMemoryDelayType01Z
-- Input     - Vector
-- Output    - Scalar
-- Delay     - Vector
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING :="";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType  := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
) IS
  VARIABLE ScheduleDataArray :
            VitalMemoryScheduleDataVectorType(0 downto 0);
  VARIABLE NumBitsPerSubword : INTEGER;
  VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
BEGIN
  PathConditionArray(0) := PathCondition;
  ScheduleDataArray(0) := ScheduleData;
  NumBitsPerSubword := ScheduleDataArray(0).NumBitsPerSubword;
  IF (OutputRetainBehavior = WordCorrupt AND
      ArcType = ParallelArc AND
      OutputRetainFlag = TRUE) THEN
    VitalMemoryUpdateInputChangeTime(
      InputChangeTimeArray,
      InputSignal,
      NumBitsPerSubword
    );
  ELSE
    VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);
  END IF;

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray, OutputRetainFlag);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #17
-- DelayType - VitalMemoryDelayType01Z
-- Input     - Vector
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
) IS
  VARIABLE NumBitsPerSubword  : INTEGER;
  VARIABLE PathConditionArray :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
BEGIN
  FOR i IN PathConditionArray'RANGE LOOP
    PathConditionArray(i) := PathCondition;
  END LOOP;

  NumBitsPerSubword :=
      ScheduleDataArray(ScheduleDataArray'LEFT).NumBitsPerSubword;
  IF (OutputRetainBehavior = WordCorrupt AND
      ArcType = ParallelArc AND
      OutputRetainFlag = TRUE) THEN
    VitalMemoryUpdateInputChangeTime(
```

 Copyright © 2001 IEEE. All rights reserved.

```
        InputChangeTimeArray,
        InputSignal,
        NumBitsPerSubword
      );
    ELSE
      VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);
    END IF;

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArray, OutputRetainFlag);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #18
  -- DelayType - VitalMemoryDelayType01Z
  -- Input     - Vector
  -- Output    - Vector
  -- Delay     - Vector
  -- Condition - Vector
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
    SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName  : IN STRING :="";
    VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
    CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
    CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
    CONSTANT PathConditionArray   : IN VitalBoolArrayT;
    CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
    CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
  ) IS
  VARIABLE NumBitsPerSubword : INTEGER;
  VARIABLE PathConditionArrayNorm :
      VitalBoolArrayT(PathConditionArray'LENGTH-1 downto 0);
  VARIABLE PathConditionArrayExp :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
  BEGIN
    NumBitsPerSubword := ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
    FOR i IN PathConditionArrayExp'RANGE LOOP
      PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
    END LOOP;

    IF (OutputRetainBehavior = WordCorrupt AND
        ArcType = ParallelArc AND
        OutputRetainFlag = TRUE) THEN
      VitalMemoryUpdateInputChangeTime(
          InputChangeTimeArray, InputSignal,
          NumBitsPerSubword);
    ELSE
      VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);
    END IF;

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArrayExp, OutputRetainFlag);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #19
  -- DelayType - VitalMemoryDelayType01XZ
  -- Input     - Scalar
  -- Output    - Scalar
  -- Delay     - Scalar
  -- Condition - Scalar
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleData      : INOUT  VitalMemoryScheduleDataType;
    SIGNAL   InputSignal       : IN STD_ULOGIC;
    CONSTANT OutputSignalName  : IN STRING :="";
    VARIABLE InputChangeTime   : INOUT TIME;
    CONSTANT PathDelay         : IN VitalDelayType01ZX;
    CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
    CONSTANT PathCondition     : IN BOOLEAN := TRUE;
    CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
  ) IS
    VARIABLE ScheduleDataArray :
           VitalMemoryScheduleDataVectorType(0 downto 0);
    VARIABLE PathDelayArray : VitalDelayArrayType01ZX(0 downto 0);
    VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
    VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
```

Copyright © 2001 IEEE. All rights reserved.

```
BEGIN
  PathConditionArray(0) := PathCondition;
  ScheduleDataArray(0) := ScheduleData;
  PathDelayArray(0) := PathDelay;
  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray, OutputRetainFlag);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #20
-- DelayType - VitalMemoryDelayType01XZ
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Scalar
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray :INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTime   : INOUT TIME;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
) IS
  VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
  VARIABLE PathConditionArray :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
BEGIN
  FOR i IN PathConditionArray'RANGE LOOP
    PathConditionArray(i) := PathCondition;
  END LOOP;

  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
    ScheduleDataArray, InputChangeTimeArray,
    OutputSignalName, PathDelayArray,
    ArcType, PathConditionArray, OutputRetainFlag);
END VitalMemoryAddPathDelay;

-- ----------------------------------------------------------------------------
-- #21
-- DelayType - VitalMemoryDelayType01XZ
-- Input     - Scalar
-- Output    - Vector
-- Delay     - Vector
-- Condition - Vector
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray :INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING :="";
  VARIABLE InputChangeTime   : INOUT TIME;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE
) IS
  VARIABLE InputChangeTimeArray : VitalTimeArrayT(0 downto 0);
  VARIABLE NumBitsPerSubword : INTEGER;
  VARIABLE PathConditionArrayNorm :
      VitalBoolArrayT(PathConditionArray'LENGTH-1 downto 0);
  VARIABLE PathConditionArrayExp :
      VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
BEGIN
  NumBitsPerSubword :=
      ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
  FOR i IN PathConditionArrayExp'RANGE LOOP
    PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
  END LOOP;

  VitalMemoryUpdateInputChangeTime(InputChangeTime, InputSignal);
  InputChangeTimeArray(0) := InputChangeTime;

  VitalMemorySelectDelay(
```

 Copyright © 2001 IEEE. All rights reserved.

```
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArrayExp, OutputRetainFlag);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #22
  -- DelayType - VitalMemoryDelayType01XZ
  -- Input     - Vector
  -- Output    - Scalar
  -- Delay     - Vector
  -- Condition - Scalar
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleData         : INOUT  VitalMemoryScheduleDataType;
    SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName     : IN STRING :="";
    VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
    CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
    CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
    CONSTANT PathCondition        : IN BOOLEAN := TRUE;
    CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
    CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
  ) IS
    VARIABLE ScheduleDataArray :
      VitalMemoryScheduleDataVectorType(0 downto 0);
    VARIABLE NumBitsPerSubword : INTEGER;
    VARIABLE PathConditionArray : VitalBoolArrayT(0 downto 0);
  BEGIN
    PathConditionArray(0) := PathCondition;
    ScheduleDataArray(0) := ScheduleData;
    NumBitsPerSubword :=
        ScheduleDataArray(ScheduleDataArray'LEFT).NumBitsPerSubword;
    IF (OutputRetainBehavior = WordCorrupt AND
        ArcType = ParallelArc AND
        OutputRetainFlag = TRUE) THEN
      VitalMemoryUpdateInputChangeTime(
        InputChangeTimeArray, InputSignal,
        NumBitsPerSubword);
    ELSE
      VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);
    END IF;

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArray, OutputRetainFlag);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #23
  -- DelayType - VitalMemoryDelayType01XZ
  -- Input     - Vector
  -- Output    - Vector
  -- Delay     - Vector
  -- Condition - Scalar
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
    SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName  : IN STRING :="";
    VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
    CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
    CONSTANT ArcType              : IN VitalMemoryArcType  := CrossArc;
    CONSTANT PathCondition        : IN BOOLEAN := TRUE;
    CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
    CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
  ) IS
    VARIABLE NumBitsPerSubword  : INTEGER;
    VARIABLE PathConditionArray :
        VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
  BEGIN
    FOR i IN PathConditionArray'RANGE LOOP
      PathConditionArray(i) := PathCondition;
    END LOOP;

    NumBitsPerSubword :=
        ScheduleDataArray(ScheduleDataArray'LEFT).NumBitsPerSubword;
    IF (OutputRetainBehavior = WordCorrupt AND
        ArcType = ParallelArc AND
        OutputRetainFlag = TRUE) THEN
      VitalMemoryUpdateInputChangeTime(
        InputChangeTimeArray, InputSignal,
```

Copyright © 2001 IEEE. All rights reserved.

```
        NumBitsPerSubword);
    ELSE
      VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);
    END IF;

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArray, OutputRetainFlag);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- #24
  -- DelayType - VitalMemoryDelayType01XZ
  -- Input     - Vector
  -- Output    - Vector
  -- Delay     - Vector
  -- Condition - Vector
  PROCEDURE VitalMemoryAddPathDelay (
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
    SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName  : IN STRING :="";
    VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
    CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
    CONSTANT ArcType              : IN VitalMemoryArcType  := CrossArc;
    CONSTANT PathConditionArray   : IN VitalBoolArrayT;
    CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
    CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt
  ) IS
    VARIABLE NumBitsPerSubword : INTEGER;
    VARIABLE PathConditionArrayNorm :
        VitalBoolArrayT(PathConditionArray'LENGTH-1 downto 0);
    VARIABLE PathConditionArrayExp :
        VitalBoolArrayT(ScheduleDataArray'LENGTH-1 downto 0);
  BEGIN
    NumBitsPerSubword :=
        ScheduleDataArray(ScheduleDataArray'RIGHT).NumBitsPerSubword;
    FOR i IN PathConditionArrayExp'RANGE LOOP
      PathConditionArrayExp(i) := PathConditionArrayNorm(i/NumBitsPerSubword);
    END LOOP;

    IF (OutputRetainBehavior = WordCorrupt AND
        ArcType = ParallelArc AND
        OutputRetainFlag = TRUE) THEN
      VitalMemoryUpdateInputChangeTime(
        InputChangeTimeArray, InputSignal,
        NumBitsPerSubword);
    ELSE
      VitalMemoryUpdateInputChangeTime(InputChangeTimeArray, InputSignal);
    END IF;

    VitalMemorySelectDelay(
      ScheduleDataArray, InputChangeTimeArray,
      OutputSignalName, PathDelayArray,
      ArcType, PathConditionArrayExp, OutputRetainFlag);
  END VitalMemoryAddPathDelay;

  -- ----------------------------------------------------------------------------
  -- Procedure:   VitalMemorySchedulePathDelay
  -- Description: Schedule Output after Propagation Delay selected
  --              by checking all the paths added thru'
  --              VitalMemoryAddPathDelay.
  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemorySchedulePathDelay (
    SIGNAL   OutSignal        : OUT STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName : IN STRING :="";
    CONSTANT PortFlag         : IN VitalPortFlagType := VitalDefaultPortFlag;
    CONSTANT OutputMap        : IN VitalOutputMapType:= VitalDefaultOutputMap;
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType
  ) IS
    VARIABLE Age               : TIME;
    VARIABLE PropDelay         : TIME;
    VARIABLE RetainDelay       : TIME;
    VARIABLE Data              : STD_ULOGIC;
  BEGIN
    IF (PortFlag.OutputDisable /= TRUE) THEN
      FOR i IN ScheduleDataArray'RANGE LOOP
        PropDelay := ScheduleDataArray(i).PropDelay;
        RetainDelay := ScheduleDataArray(i).OutputRetainDelay;

        NEXT WHEN PropDelay = TIME'HIGH;
```

 Copyright © 2001 IEEE. All rights reserved.

```
        Age := ScheduleDataArray(i).InputAge;
        Data := ScheduleDataArray(i).OutputData;

        IF (Age < RetainDelay and RetainDelay < PropDelay) THEN
          OutSignal(i) <= TRANSPORT 'X' AFTER (RetainDelay - Age);
        END IF;

        IF (Age <= PropDelay) THEN
          OutSignal(i)<= TRANSPORT OutputMap(Data)AFTER (PropDelay-Age);
          ScheduleDataArray(i).ScheduleValue := Data;
          ScheduleDataArray(i).ScheduleTime := NOW + PropDelay - Age;
        END IF;
      END LOOP;
    END IF;

    -- for debug purpose
    PrintScheduleDataArray(ScheduleDataArray);

    -- for debug purpose
    ScheduleDebugMsg;
  END VitalMemorySchedulePathDelay;

  -- ----------------------------------------------------------------------------
  -- Procedure:   VitalMemorySchedulePathDelay
  -- Description: Schedule Output after Propagation Delay selected
  --              by checking all the paths added thru'
  --              VitalMemoryAddPathDelay.
  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemorySchedulePathDelay (
    SIGNAL   OutSignal        : OUT STD_LOGIC_VECTOR;
    CONSTANT OutputSignalName : IN STRING :="";
    CONSTANT PortFlag         : IN VitalPortFlagVectorType;
    CONSTANT OutputMap        : IN VitalOutputMapType:= VitalDefaultOutputMap;
    VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType
  ) IS
    VARIABLE Age               : TIME;
    VARIABLE PropDelay         : TIME;
    VARIABLE RetainDelay       : TIME;
    VARIABLE Data              : STD_ULOGIC;
    VARIABLE ExpandedPortFlag  :
        VitalPortFlagVectorType(ScheduleDataArray'RANGE);
    VARIABLE NumBitsPerSubword : INTEGER;
  BEGIN
    NumBitsPerSubword :=
        ScheduleDataArray(ScheduleDataArray'LEFT).NumBitsPerSubword;
    VitalMemoryExpandPortFlag( PortFlag, NumBitsPerSubword, ExpandedPortFlag );
    FOR i IN ScheduleDataArray'RANGE LOOP
      NEXT WHEN ExpandedPortFlag(i).OutputDisable = TRUE;

      PropDelay := ScheduleDataArray(i).PropDelay;
      RetainDelay := ScheduleDataArray(i).OutputRetainDelay;

      NEXT WHEN PropDelay = TIME'HIGH;

      Age  := ScheduleDataArray(i).InputAge;
      Data := ScheduleDataArray(i).OutputData;

      IF (Age < RetainDelay and RetainDelay < PropDelay) THEN
        OutSignal(i) <= TRANSPORT 'X' AFTER (RetainDelay - Age);
      END IF;

      IF (Age <= PropDelay) THEN
        OutSignal(i)<= TRANSPORT OutputMap(Data)AFTER (PropDelay-Age);
        ScheduleDataArray(i).ScheduleValue := Data;
        ScheduleDataArray(i).ScheduleTime := NOW + PropDelay - Age;
      END IF;
    END LOOP;

    -- for debug purpose
    PrintScheduleDataArray(ScheduleDataArray);

    -- for debug purpose
    ScheduleDebugMsg;
  END VitalMemorySchedulePathDelay;

  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemorySchedulePathDelay (
    SIGNAL   OutSignal       : OUT STD_ULOGIC;
    CONSTANT OutputSignalName: IN STRING :="";
    CONSTANT PortFlag        : IN VitalPortFlagType := VitalDefaultPortFlag;
```

Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT OutputMap       : IN VitalOutputMapType := VitalDefaultOutputMap;
    VARIABLE ScheduleData    : INOUT VitalMemoryScheduleDataType
  ) IS
    VARIABLE Age               : TIME;
    VARIABLE PropDelay         : TIME;
    VARIABLE RetainDelay       : TIME;
    VARIABLE Data              : STD_ULOGIC;
    VARIABLE ScheduleDataArray : VitalMemoryScheduleDataVectorType (0 downto 0);
  BEGIN
    IF (PortFlag.OutputDisable /= TRUE) THEN
      ScheduledataArray(0) := ScheduleData;
      PropDelay := ScheduleDataArray(0).PropDelay;
      RetainDelay := ScheduleDataArray(0).OutputRetainDelay;
      Age := ScheduleDataArray(0).InputAge;
      Data := ScheduleDataArray(0).OutputData;

      IF (Age < RetainDelay and RetainDelay < PropDelay) THEN
        OutSignal <= TRANSPORT 'X' AFTER (RetainDelay - Age);
      END IF;

      IF (Age <= PropDelay and PropDelay /= TIME'HIGH) THEN
        OutSignal <= TRANSPORT OutputMap(Data)  AFTER (PropDelay - Age);
        ScheduleDataArray(0).ScheduleValue := Data;
        ScheduleDataArray(0).ScheduleTime := NOW + PropDelay - Age;
      END IF;
    END IF;

    -- for debug purpose
    PrintScheduleDataArray(ScheduleDataArray);

    -- for debug purpose
    ScheduleDebugMsg;

  END VitalMemorySchedulePathDelay;

  -- ----------------------------------------------------------------------------
  -- Procedure  : InternalTimingCheck
  -- ----------------------------------------------------------------------------
  PROCEDURE InternalTimingCheck (
    CONSTANT TestSignal    : IN     std_ulogic;
    CONSTANT RefSignal     : IN     std_ulogic;
    CONSTANT TestDelay     : IN     TIME := 0 ns;
    CONSTANT RefDelay      : IN     TIME := 0 ns;
    CONSTANT SetupHigh     : IN     TIME := 0 ns;
    CONSTANT SetupLow      : IN     TIME := 0 ns;
    CONSTANT HoldHigh      : IN     TIME := 0 ns;
    CONSTANT HoldLow       : IN     TIME := 0 ns;
    VARIABLE RefTime       : IN     TIME;
    VARIABLE RefEdge       : IN     BOOLEAN;
    VARIABLE TestTime      : IN     TIME;
    VARIABLE TestEvent     : IN     BOOLEAN;
    VARIABLE SetupEn       : INOUT  BOOLEAN;
    VARIABLE HoldEn        : INOUT  BOOLEAN;
    VARIABLE CheckInfo     : INOUT  CheckInfoType;
    CONSTANT MsgOn         : IN     BOOLEAN
  ) IS
    VARIABLE bias           : TIME;
    VARIABLE actualObsTime  : TIME;
    VARIABLE BC             : TIME;
    VARIABLE Message        :LINE;
  BEGIN
    -- Check SETUP constraint
    IF (RefEdge) THEN
      IF (SetupEn) THEN
        CheckInfo.ObsTime   := RefTime - TestTime;
        CheckInfo.State     := To_X01(TestSignal);
        CASE CheckInfo.State IS
          WHEN '0' =>
            CheckInfo.ExpTime := SetupLow;
            -- start of new code IR245-246
            BC := HoldHigh;
            -- end of new code IR245-246
          WHEN '1' =>
            CheckInfo.ExpTime := SetupHigh;
            -- start of new code IR245-246
            BC := HoldLow;
            -- end of new code IR245-246
          WHEN 'X' =>
            CheckInfo.ExpTime := Maximum(SetupHigh,SetupLow);
            -- start of new code IR245-246
            BC := Maximum(HoldHigh,HoldLow);
```

 Copyright © 2001 IEEE. All rights reserved.

```
          -- end of new code IR245-246
      END CASE;
      -- added the second condition for IR 245-246
      CheckInfo.Violation :=
          ((CheckInfo.ObsTime < CheckInfo.ExpTime)
            AND ( NOT ((CheckInfo.ObsTime = BC) and (BC = 0 ns))));
      -- start of new code IR245-246
      IF (CheckInfo.ExpTime = 0 ns) THEN
        CheckInfo.CheckKind := HoldCheck;
      ELSE
        CheckInfo.CheckKind := SetupCheck;
      END IF;
      -- end of new code IR245-246
      SetupEn  := FALSE;
    ELSE
      CheckInfo.Violation := FALSE;
    END IF;

  -- Check HOLD constraint
  ELSIF (TestEvent) THEN
    IF HoldEn THEN
      CheckInfo.ObsTime := TestTime - RefTime;
      CheckInfo.State := To_X01(TestSignal);
      CASE CheckInfo.State IS
        WHEN '0' =>
          CheckInfo.ExpTime := HoldHigh;
          -- new code for unnamed IR
          CheckInfo.State := '1';
          -- start of new code IR245-246
          BC := SetupLow;
          -- end of new code IR245-246
        WHEN '1' =>
          CheckInfo.ExpTime := HoldLow;
          -- new code for unnamed IR
          CheckInfo.State := '0';
          -- start of new code IR245-246
          BC := SetupHigh;
          -- end of new code IR245-246
        WHEN 'X' =>
          CheckInfo.ExpTime := Maximum(HoldHigh,HoldLow);
          -- start of new code IR245-246
          BC := Maximum(SetupHigh,SetupLow);
          -- end of new code IR245-246
      END CASE;
      -- added the second condition for IR 245-246
      CheckInfo.Violation :=
        ((CheckInfo.ObsTime < CheckInfo.ExpTime)
          AND ( NOT ((CheckInfo.ObsTime = BC) and (BC = 0 ns))));
      -- start of new code IR245-246
      IF (CheckInfo.ExpTime = 0 ns) THEN
        CheckInfo.CheckKind := SetupCheck;
      ELSE
        CheckInfo.CheckKind := HoldCheck;
      END IF;
      -- end of new code IR245-246
      HoldEn := NOT CheckInfo.Violation;
    ELSE
      CheckInfo.Violation := FALSE;
    END IF;
  ELSE
    CheckInfo.Violation := FALSE;
  END IF;

  -- Adjust report values to account for internal model delays
  -- Note: TestDelay, RefDelay, TestTime, RefTime are non-negative
  -- Note: bias may be negative or positive
  IF MsgOn AND CheckInfo.Violation THEN
    -- modified the code for correct reporting of violation in case of
    -- order of signals being reversed because of internal delays
    -- new variable
    actualObsTime := (TestTime-TestDelay)-(RefTime-RefDelay);
    bias := TestDelay - RefDelay;
    IF (actualObsTime < 0 ns) THEN -- It should be a setup check
      IF ( CheckInfo.CheckKind = HoldCheck) THEN
        CheckInfo.CheckKind := SetupCheck;
        CASE CheckInfo.State IS
          WHEN '0' => CheckInfo.ExpTime := SetupLow;
          WHEN '1' => CheckInfo.ExpTime := SetupHigh;
          WHEN 'X' => CheckInfo.ExpTime := Maximum(SetupHigh,SetupLow);
        END CASE;
      END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
      CheckInfo.ObsTime := -actualObsTime;
      CheckInfo.ExpTime := CheckInfo.ExpTime + bias;
      CheckInfo.DetTime := RefTime - RefDelay;
    ELSE -- It should be a hold check
      IF (CheckInfo.CheckKind = SetupCheck) THEN
        CheckInfo.CheckKind :=  HoldCheck;
        CASE CheckInfo.State IS
          WHEN '0' =>
            CheckInfo.ExpTime := HoldHigh;
            CheckInfo.State   := '1';
          WHEN '1' =>
            CheckInfo.ExpTime := HoldLow;
            CheckInfo.State   := '0';
          WHEN 'X' =>
            CheckInfo.ExpTime := Maximum(HoldHigh,HoldLow);
        END CASE;
      END IF;
      CheckInfo.ObsTime := actualObsTime;
      CheckInfo.ExpTime := CheckInfo.ExpTime - bias;
      CheckInfo.DetTime := TestTime - TestDelay;
    END IF;
  END IF;
END InternalTimingCheck;


-- ------------------------------------------------------------------------------
-- Setup and Hold Time Check Routine
-- ------------------------------------------------------------------------------
PROCEDURE TimingArrayIndex (
  SIGNAL InputSignal      : IN Std_logic_vector;
  CONSTANT ArrayIndexNorm : IN INTEGER;
  VARIABLE Index          : OUT INTEGER
) IS
BEGIN
  IF (InputSignal'LEFT > InputSignal'RIGHT) THEN
    Index  := ArrayIndexNorm + InputSignal'RIGHT;
  ELSE
    Index  := InputSignal'RIGHT - ArrayIndexNorm;
  END IF;
END TimingArrayIndex;

-- ------------------------------------------------------------------------------
PROCEDURE VitalMemoryReportViolation (
  CONSTANT TestSignalName : IN STRING := "";
  CONSTANT RefSignalName  : IN STRING := "";
  CONSTANT HeaderMsg      : IN STRING := " ";
  CONSTANT CheckInfo      : IN CheckInfoType;
  CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
) IS
  VARIABLE Message : LINE;
BEGIN
  IF (NOT CheckInfo.Violation) THEN
    RETURN;
  END IF;
  Write ( Message, HeaderMsg );
  CASE CheckInfo.CheckKind IS
    WHEN    SetupCheck => Write ( Message, STRING'(" SETUP ")      );
    WHEN     HoldCheck => Write ( Message, STRING'(" HOLD ")       );
    WHEN RecoveryCheck => Write ( Message, STRING'(" RECOVERY ")   );
    WHEN  RemovalCheck => Write ( Message, STRING'(" REMOVAL ")    );
    WHEN PulseWidCheck => Write ( Message, STRING'(" PULSE WIDTH "));
    WHEN   PeriodCheck => Write ( Message, STRING'(" PERIOD ")     );
  END CASE;
  Write ( Message, HiLoStr(CheckInfo.State) );
  Write ( Message, STRING'(" VIOLATION ON ") );
  Write ( Message, TestSignalName );
  IF (RefSignalName'LENGTH > 0) THEN
    Write ( Message, STRING'(" WITH RESPECT TO ") );
    Write ( Message, RefSignalName );
  END IF;
  Write ( Message, ';' & LF );
  Write ( Message, STRING'("  Expected := ")  );
  Write ( Message, CheckInfo.ExpTime);
  Write ( Message, STRING'("; Observed := ")  );
  Write ( Message, CheckInfo.ObsTime);
  Write ( Message, STRING'("; At : ")         );
  Write ( Message, CheckInfo.DetTime);
  ASSERT FALSE REPORT Message.ALL SEVERITY MsgSeverity;
  DEALLOCATE (Message);
END VitalMemoryReportViolation;
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- ----------------------------------------------------------------------------
PROCEDURE VitalMemoryReportViolation (
  CONSTANT TestSignalName : IN STRING := "";
  CONSTANT RefSignalName  : IN STRING := "";
  CONSTANT TestArrayIndex : IN INTEGER;
  CONSTANT RefArrayIndex  : IN INTEGER;
  SIGNAL TestSignal       : IN std_logic_vector;
  SIGNAL RefSignal        : IN std_logic_vector;
  CONSTANT HeaderMsg      : IN STRING := " ";
  CONSTANT CheckInfo      : IN CheckInfoType;
  CONSTANT MsgFormat      : IN VitalMemoryMsgFormatType;
  CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
) IS
  VARIABLE Message : LINE;
  VARIABLE i, j : INTEGER;
BEGIN
  IF (NOT CheckInfo.Violation) THEN
    RETURN;
  END IF;

  Write ( Message, HeaderMsg );
  CASE CheckInfo.CheckKind IS
    WHEN    SetupCheck => Write ( Message, STRING'(" SETUP ")      );
    WHEN     HoldCheck => Write ( Message, STRING'(" HOLD ")       );
    WHEN PulseWidCheck => Write ( Message, STRING'(" PULSE WIDTH "));
    WHEN   PeriodCheck => Write ( Message, STRING'(" PERIOD ")     );
    WHEN OTHERS        => Write ( Message, STRING'(" UNKNOWN ")     );
  END CASE;
  Write ( Message, HiLoStr(CheckInfo.State) );
  Write ( Message, STRING'(" VIOLATION ON ") );
  Write ( Message, TestSignalName );
  TimingArrayIndex(TestSignal, TestArrayIndex, i);
  CASE MsgFormat IS
    WHEN Scalar =>
      NULL;
    WHEN VectorEnum =>
      Write ( Message, '_');
      Write ( Message, i);
    WHEN Vector =>
      Write ( Message, '(');
      Write ( Message, i);
      Write ( Message, ')');
  END CASE;

  IF (RefSignalName'LENGTH > 0) THEN
    Write ( Message, STRING'(" WITH RESPECT TO ") );
    Write ( Message, RefSignalName );
  END IF;

  IF(RefSignal'LENGTH > 0) THEN
    TimingArrayIndex(RefSignal, RefArrayIndex, j);
    CASE MsgFormat IS
      WHEN Scalar =>
        NULL;
      WHEN VectorEnum =>
        Write ( Message, '_');
        Write ( Message, j);
      WHEN Vector =>
        Write ( Message, '(');
        Write ( Message, j);
        Write ( Message, ')');
    END CASE;
  END IF;

  Write ( Message, ';' & LF );
  Write ( Message, STRING'("  Expected := ")  );
  Write ( Message, CheckInfo.ExpTime);
  Write ( Message, STRING'("; Observed := ")  );
  Write ( Message, CheckInfo.ObsTime);
  Write ( Message, STRING'("; At : ")         );
  Write ( Message, CheckInfo.DetTime);

  ASSERT FALSE REPORT Message.ALL SEVERITY MsgSeverity;

  DEALLOCATE (Message);
END VitalMemoryReportViolation;

-- ----------------------------------------------------------------------------
PROCEDURE VitalMemoryReportViolation (
  CONSTANT TestSignalName : IN STRING := "";
  CONSTANT RefSignalName  : IN STRING := "";
```

Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT TestArrayIndex : IN INTEGER;
    CONSTANT HeaderMsg      : IN STRING := " ";
    CONSTANT CheckInfo      : IN CheckInfoType;
    CONSTANT MsgFormat      : IN VitalMemoryMsgFormatType;
    CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
  ) IS
    VARIABLE Message : LINE;
  BEGIN
    IF (NOT CheckInfo.Violation) THEN
      RETURN;
    END IF;

    Write ( Message, HeaderMsg );
    CASE CheckInfo.CheckKind IS
      WHEN    SetupCheck => Write ( Message, STRING'(" SETUP ")      );
      WHEN     HoldCheck => Write ( Message, STRING'(" HOLD ")       );
      WHEN PulseWidCheck => Write ( Message, STRING'(" PULSE WIDTH "));
      WHEN   PeriodCheck => Write ( Message, STRING'(" PERIOD ")     );
      WHEN        OTHERS => Write ( Message, STRING'(" UNKNOWN ")     );
    END CASE;

    Write ( Message, HiLoStr(CheckInfo.State) );
    Write ( Message, STRING'(" VIOLATION ON ") );
    Write ( Message, TestSignalName );

    CASE MsgFormat IS
      WHEN Scalar =>
        NULL;
      WHEN VectorEnum =>
        Write ( Message, '_');
        Write ( Message, TestArrayIndex);
      WHEN Vector =>
        Write ( Message, '(');
        Write ( Message, TestArrayIndex);
        Write ( Message, ')');
    END CASE;

    IF (RefSignalName'LENGTH > 0) THEN
      Write ( Message, STRING'(" WITH RESPECT TO ") );
      Write ( Message, RefSignalName );
    END IF;

    Write ( Message, ';' & LF );
    Write ( Message, STRING'("  Expected := ")  );
    Write ( Message, CheckInfo.ExpTime);
    Write ( Message, STRING'("; Observed := ")  );
    Write ( Message, CheckInfo.ObsTime);
    Write ( Message, STRING'("; At : ")         );
    Write ( Message, CheckInfo.DetTime);

    ASSERT FALSE REPORT Message.ALL SEVERITY MsgSeverity;

    DEALLOCATE (Message);
  END VitalMemoryReportViolation;

  -- ------------------------------------------------------------------------------
  FUNCTION VitalMemoryTimingDataInit
  RETURN VitalMemoryTimingDataType IS
  BEGIN
    RETURN (FALSE, 'X', 0 ns, FALSE, 'X', 0 ns, FALSE,
            NULL, NULL, NULL, NULL, NULL, NULL);
  END;

  -- ------------------------------------------------------------------------------
  -- Procedure:   VitalSetupHoldCheck
  -- ------------------------------------------------------------------------------
  PROCEDURE VitalMemorySetupHoldCheck (
    VARIABLE Violation     : OUT    X01ArrayT;
    VARIABLE TimingData    : INOUT  VitalMemoryTimingDataType;
    SIGNAL   TestSignal    : IN     std_ulogic;
    CONSTANT TestSignalName: IN     STRING := "";
    CONSTANT TestDelay     : IN     TIME := 0 ns;
    SIGNAL   RefSignal     : IN     std_ulogic;
    CONSTANT RefSignalName : IN     STRING := "";
    CONSTANT RefDelay      : IN     TIME := 0 ns;
    CONSTANT SetupHigh     : IN     VitalDelayType;
    CONSTANT SetupLow      : IN     VitalDelayType;
    CONSTANT HoldHigh      : IN     VitalDelayType;
    CONSTANT HoldLow       : IN     VitalDelayType;
    CONSTANT CheckEnabled  : IN     VitalBoolArrayT;
    CONSTANT RefTransition : IN     VitalEdgeSymbolType;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT HeaderMsg     : IN     STRING := " ";
    CONSTANT XOn           : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
    --IR252 3/23/98
    CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;
    CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;
    CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;
    CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE
  ) IS
    VARIABLE CheckInfo       : CheckInfoType;
    VARIABLE CheckEnScalar   : BOOLEAN := FALSE;
    VARIABLE ViolationInt    : X01ArrayT(CheckEnabled'RANGE);
    VARIABLE RefEdge         : BOOLEAN;
    VARIABLE TestEvent       : BOOLEAN;
    VARIABLE TestDly         : TIME := Maximum(0 ns, TestDelay);
    VARIABLE RefDly          : TIME := Maximum(0 ns, RefDelay);
    VARIABLE bias            : TIME;
  BEGIN

    -- Initialization of working area.
    IF (TimingData.NotFirstFlag = FALSE) THEN
      TimingData.TestLast     := To_X01(TestSignal);
      TimingData.RefLast      := To_X01(RefSignal);
      TimingData.NotFirstFlag := TRUE;
    END IF;

    -- Detect reference edges and record the time of the last edge
    RefEdge := EdgeSymbolMatch(TimingData.RefLast, To_X01(RefSignal),
                 RefTransition);
    TimingData.RefLast := To_X01(RefSignal);
    IF (RefEdge) THEN
      TimingData.RefTime     := NOW;
      --TimingData.HoldEnA.all := (TestSignal'RANGE=>TRUE);
      --IR252 3/23/98
      TimingData.SetupEn := TimingData.SetupEn AND EnableSetupOnRef;
      TimingData.HoldEn  := EnableHoldOnRef;
    END IF;

    -- Detect test (data) changes and record the time of the last change
    TestEvent := TimingData.TestLast /= To_X01Z(TestSignal);
    TimingData.TestLast := To_X01Z(TestSignal);
    IF TestEvent THEN
      TimingData.SetupEn  := EnableSetupOnTest ; --IR252 3/23/98
      TimingData.HoldEn := TimingData.HoldEn AND EnableHoldOnTest ;
                                                --IR252 3/23/98
      TimingData.TestTime := NOW;
    END IF;

    FOR i IN CheckEnabled'RANGE LOOP
      IF CheckEnabled(i) = TRUE THEN
        CheckEnScalar := TRUE;
      END IF;
      ViolationInt(i) := '0';
    END LOOP;

    IF (CheckEnScalar) THEN
      InternalTimingCheck (
        TestSignal   => TestSignal,
        RefSignal    => RefSignal,
        TestDelay    => TestDly,
        RefDelay     => RefDly,
        SetupHigh    => SetupHigh,
        SetupLow     => SetupLow,
        HoldHigh     => HoldHigh,
        HoldLow      => HoldLow,
        RefTime      => TimingData.RefTime,
        RefEdge      => RefEdge,
        TestTime     => TimingData.TestTime,
        TestEvent    => TestEvent,
        SetupEn      => TimingData.SetupEn,
        HoldEn       => TimingData.HoldEn,
        CheckInfo    => CheckInfo,
        MsgOn        => MsgOn
      );

      -- Report any detected violations and set return violation flag
      IF CheckInfo.Violation THEN
        IF (MsgOn) THEN
          VitalMemoryReportViolation (TestSignalName, RefSignalName,
            HeaderMsg, CheckInfo, MsgSeverity );
```

Copyright © 2001 IEEE. All rights reserved.

```
      END IF;
      IF (XOn) THEN
        FOR i IN CheckEnabled'RANGE LOOP
          IF CheckEnabled(i) = TRUE THEN
              ViolationInt(i) := 'X';
          END IF;
        END LOOP;
      END IF;
    END IF;
  END IF;
  Violation := ViolationInt;
END VitalMemorySetupHoldCheck;

-- ---------------------------------------------------------------------------
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation      : OUT    X01ArrayT;
  VARIABLE TimingData     : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal     : IN     std_logic_vector;
  CONSTANT TestSignalName: IN      STRING := "";
  CONSTANT TestDelay      : IN     VitalDelayArraytype;
  SIGNAL   RefSignal      : IN     std_ulogic;
  CONSTANT RefSignalName  : IN     STRING := "";
  CONSTANT RefDelay       : IN     TIME := 0 ns;
  CONSTANT SetupHigh      : IN     VitalDelayArraytype;
  CONSTANT SetupLow       : IN     VitalDelayArraytype;
  CONSTANT HoldHigh       : IN     VitalDelayArraytype;
  CONSTANT HoldLow        : IN     VitalDelayArraytype;
  CONSTANT CheckEnabled   : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition  : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg      : IN     STRING := " ";
  CONSTANT XOn            : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn          : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat      : IN     VitalMemoryMsgFormatType;
  --IR252 3/23/98
  CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE
) IS
  VARIABLE CheckInfo      : CheckInfoType;
  VARIABLE RefEdge        : BOOLEAN;
  VARIABLE TestEvent      : VitalBoolArrayT(TestSignal'RANGE);
  VARIABLE TestDly        : TIME;
  VARIABLE RefDly         : TIME := Maximum(0 ns, RefDelay);
  VARIABLE bias           : TIME;
BEGIN

  -- Initialization of working area.
  IF (TimingData.NotFirstFlag = FALSE) THEN
    TimingData.TestLastA := NEW std_logic_vector(TestSignal'RANGE);
    TimingData.TestTimeA := NEW VitalTimeArrayT(TestSignal'RANGE);
    TimingData.HoldEnA   := NEW VitalBoolArrayT(TestSignal'RANGE);
    TimingData.SetupEnA  := NEW VitalBoolArrayT(TestSignal'RANGE);
    FOR i IN TestSignal'RANGE LOOP
      TimingData.TestLastA(i) := To_X01(TestSignal(i));
    END LOOP;
    TimingData.RefLast := To_X01(RefSignal);
    TimingData.NotFirstFlag := TRUE;
  END IF;

  -- Detect reference edges and record the time of the last edge
  RefEdge := EdgeSymbolMatch(TimingData.RefLast, To_X01(RefSignal),
             RefTransition);
  TimingData.RefLast := To_X01(RefSignal);
  IF (RefEdge) THEN
    TimingData.RefTime     := NOW;
    --TimingData.HoldEnA.all := (TestSignal'RANGE=>TRUE);
    --IR252 3/23/98
    FOR i IN TestSignal'RANGE LOOP
      TimingData.SetupEnA(i)
        := TimingData.SetupEnA(i) AND EnableSetupOnRef;
      TimingData.HoldEnA(i)  := EnableHoldOnRef;
    END LOOP;
  END IF;

  -- Detect test (data) changes and record the time of the last change
  FOR i IN TestSignal'RANGE LOOP
    TestEvent(i) := TimingData.TestLastA(i) /= To_X01Z(TestSignal(i));
    TimingData.TestLastA(i) := To_X01Z(TestSignal(i));
    IF TestEvent(i) THEN
```

 Copyright © 2001 IEEE. All rights reserved.

```
        TimingData.SetupEnA(i)  := EnableSetupOnTest ; --IR252 3/23/98
        TimingData.HoldEnA(i) := TimingData.HoldEnA(i) AND EnableHoldOnTest ;
                                              --IR252 3/23/98
        TimingData.TestTimeA(i) := NOW;
        --TimingData.SetupEnA(i)  := TRUE;
        TimingData.TestTime := NOW;
      END IF;
    END LOOP;

    FOR i IN TestSignal'RANGE LOOP
      Violation(i) := '0';

      IF (CheckEnabled) THEN
        TestDly := Maximum(0 ns, TestDelay(i));
        InternalTimingCheck (
          TestSignal   => TestSignal(i),
          RefSignal    => RefSignal,
          TestDelay    => TestDly,
          RefDelay     => RefDly,
          SetupHigh    => SetupHigh(i),
          SetupLow     => SetupLow(i),
          HoldHigh     => HoldHigh(i),
          HoldLow      => HoldLow(i),
          RefTime      => TimingData.RefTime,
          RefEdge      => RefEdge,
          TestTime     => TimingData.TestTimeA(i),
          TestEvent    => TestEvent(i),
          SetupEn      => TimingData.SetupEnA(i),
          HoldEn       => TimingData.HoldEnA(i),
          CheckInfo    => CheckInfo,
          MsgOn        => MsgOn
        );

        -- Report any detected violations and set return violation flag
        IF CheckInfo.Violation THEN
          IF (MsgOn) THEN
            VitalMemoryReportViolation (TestSignalName, RefSignalName, i ,
              HeaderMsg, CheckInfo, MsgFormat, MsgSeverity );
          END IF;
          IF (XOn) THEN
            Violation(i) := 'X';
          END IF;
        END IF;
      END IF;
    END LOOP;

  END VitalMemorySetupHoldCheck;

  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemorySetupHoldCheck (
    VARIABLE Violation     : OUT    X01ArrayT;
    VARIABLE TimingData    : INOUT  VitalMemoryTimingDataType;
    SIGNAL   TestSignal    : IN     std_logic_vector;
    CONSTANT TestSignalName: IN     STRING := "";
    CONSTANT TestDelay     : IN     VitalDelayArraytype;
    SIGNAL   RefSignal     : IN     std_ulogic;
    CONSTANT RefSignalName : IN     STRING := "";
    CONSTANT RefDelay      : IN     TIME := 0 ns;
    CONSTANT SetupHigh     : IN     VitalDelayArraytype;
    CONSTANT SetupLow      : IN     VitalDelayArraytype;
    CONSTANT HoldHigh      : IN     VitalDelayArraytype;
    CONSTANT HoldLow       : IN     VitalDelayArraytype;
    CONSTANT CheckEnabled  : IN     VitalBoolArrayT;
    CONSTANT RefTransition : IN     VitalEdgeSymbolType;
    CONSTANT ArcType       : IN     VitalMemoryArcType := CrossArc;
    CONSTANT NumBitsPerSubWord : IN INTEGER := 1;
    CONSTANT HeaderMsg     : IN     STRING := " ";
    CONSTANT XOn           : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT MsgFormat     : IN     VitalMemoryMsgFormatType;
    --IR252 3/23/98
    CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;
    CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;
    CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;
    CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE
  ) IS
    VARIABLE CheckInfo       : CheckInfoType;
    VARIABLE ViolationInt    : X01ArrayT(TestSignal'RANGE);
    VARIABLE ViolationIntNorm: X01ArrayT(TestSignal'LENGTH-1 downto 0);
    VARIABLE ViolationNorm   : X01ArrayT(Violation'LENGTH-1 downto 0);
```

Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE CheckEnInt      : VitalBoolArrayT(TestSignal'RANGE);
    VARIABLE CheckEnIntNorm  : VitalBoolArrayT(TestSignal'LENGTH-1 downto 0);
    VARIABLE CheckEnScalar   : BOOLEAN;
    VARIABLE CheckEnabledNorm: VitalBoolArrayT(CheckEnabled'LENGTH-1 downto 0);
    VARIABLE RefEdge         : BOOLEAN;
    VARIABLE TestEvent       : VitalBoolArrayT(TestSignal'RANGE);
    VARIABLE TestDly         : TIME;
    VARIABLE RefDly          : TIME := Maximum(0 ns, RefDelay);
    VARIABLE bias            : TIME;
  BEGIN

    -- Initialization of working area.
    IF (TimingData.NotFirstFlag = FALSE) THEN
      TimingData.TestLastA := NEW std_logic_vector(TestSignal'RANGE);
      TimingData.TestTimeA := NEW VitalTimeArrayT(TestSignal'RANGE);
      TimingData.HoldEnA   := NEW VitalBoolArrayT(TestSignal'RANGE);
      TimingData.SetupEnA  := NEW VitalBoolArrayT(TestSignal'RANGE);
      FOR i IN TestSignal'RANGE LOOP
        TimingData.TestLastA(i) := To_X01(TestSignal(i));
      END LOOP;
      TimingData.RefLast := To_X01(RefSignal);
      TimingData.NotFirstFlag := TRUE;
    END IF;

    -- Detect reference edges and record the time of the last edge
    RefEdge := EdgeSymbolMatch(TimingData.RefLast, To_X01(RefSignal),
                 RefTransition);
    TimingData.RefLast := To_X01(RefSignal);
    IF RefEdge THEN
      TimingData.RefTime     := NOW;
      --TimingData.HoldEnA.all := (TestSignal'RANGE=>TRUE);
      --IR252 3/23/98
      FOR i IN TestSignal'RANGE LOOP
        TimingData.SetupEnA(i)
          := TimingData.SetupEnA(i) AND EnableSetupOnRef;
        TimingData.HoldEnA(i)  := EnableHoldOnRef;
      END LOOP;
    END IF;

    -- Detect test (data) changes and record the time of the last change
    FOR i IN TestSignal'RANGE LOOP
      TestEvent(i) := TimingData.TestLastA(i) /= To_X01Z(TestSignal(i));
      TimingData.TestLastA(i) := To_X01Z(TestSignal(i));
      IF TestEvent(i) THEN
        TimingData.SetupEnA(i)  := EnableSetupOnTest ; --IR252 3/23/98
        TimingData.HoldEnA(i) := TimingData.HoldEnA(i) AND EnableHoldOnTest ;
                                                     --IR252 3/23/98
        TimingData.TestTimeA(i) := NOW;
        --TimingData.SetupEnA(i)  := TRUE;
        TimingData.TestTime := NOW;
      END IF;
    END LOOP;

    IF ArcType = CrossArc THEN
      CheckEnScalar := FALSE;
      FOR i IN CheckEnabled'RANGE LOOP
        IF CheckEnabled(i) = TRUE THEN
          CheckEnScalar := TRUE;
        END IF;
      END LOOP;
      FOR i IN CheckEnInt'RANGE LOOP
        CheckEnInt(i) := CheckEnScalar;
      END LOOP;
    ELSE
      FOR i IN CheckEnIntNorm'RANGE LOOP
        CheckEnIntNorm(i) := CheckEnabledNorm(i / NumBitsPerSubWord );
      END LOOP;
      CheckEnInt := CheckEnIntNorm;
    END IF;

    FOR i IN TestSignal'RANGE LOOP
      ViolationInt(i) := '0';

      IF (CheckEnInt(i)) THEN
        TestDly := Maximum(0 ns, TestDelay(i));
        InternalTimingCheck (
          TestSignal   => TestSignal(i),
          RefSignal    => RefSignal,
          TestDelay    => TestDly,
          RefDelay     => RefDly,
          SetupHigh    => SetupHigh(i),
```

 Copyright © 2001 IEEE. All rights reserved.

```
        SetupLow     => SetupLow(i),
        HoldHigh     => HoldHigh(i),
        HoldLow      => HoldLow(i),
        RefTime      => TimingData.RefTime,
        RefEdge      => RefEdge,
        TestTime     => TimingData.TestTimeA(i),
        TestEvent    => TestEvent(i),
        SetupEn      => TimingData.SetupEnA(i),
        HoldEn       => TimingData.HoldEnA(i),
        CheckInfo    => CheckInfo,
        MsgOn        => MsgOn
      );

      -- Report any detected violations and set return violation flag
      IF CheckInfo.Violation THEN
        IF (MsgOn) THEN
          VitalMemoryReportViolation (TestSignalName, RefSignalName, i ,
            HeaderMsg, CheckInfo, MsgFormat, MsgSeverity );
        END IF;
        IF (XOn) THEN
          ViolationInt(i) := 'X';
        END IF;
      END IF;
    END IF;
  END LOOP;

  IF (ViolationInt'LENGTH = Violation'LENGTH) THEN
    Violation := ViolationInt;
  ELSE
    ViolationIntNorm := ViolationInt;
    FOR i IN ViolationNorm'RANGE LOOP
      ViolationNorm(i) := '0';
    END LOOP;
    FOR i IN ViolationIntNorm'RANGE LOOP
      IF (ViolationIntNorm(i) = 'X') THEN
        ViolationNorm(i / NumBitsPerSubWord) := 'X';
      END IF;
    END LOOP;
    Violation := ViolationNorm;
  END IF;

END VitalMemorySetupHoldCheck;

-- ------------------------------------------------------------------------------
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation     : OUT    X01ArrayT;
  VARIABLE TimingData    : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal    : IN     std_logic_vector;
  CONSTANT TestSignalName: IN     STRING := "";
  CONSTANT TestDelay     : IN     VitalDelayArraytype;
  SIGNAL   RefSignal     : IN     std_logic_vector;
  CONSTANT RefSignalName : IN     STRING := "";
  CONSTANT RefDelay      : IN     VitalDelayArraytype;
  CONSTANT SetupHigh     : IN     VitalDelayArraytype;
  CONSTANT SetupLow      : IN     VitalDelayArraytype;
  CONSTANT HoldHigh      : IN     VitalDelayArraytype;
  CONSTANT HoldLow       : IN     VitalDelayArraytype;
  CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition : IN     VitalEdgeSymbolType;
  CONSTANT ArcType       : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN INTEGER := 1;
  CONSTANT HeaderMsg     : IN     STRING := " ";
  CONSTANT XOn           : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat     : IN     VitalMemoryMsgFormatType;
  --IR252 3/23/98
  CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE
) IS
  VARIABLE CheckInfo       : CheckInfoType;
  VARIABLE RefEdge         : VitalBoolArrayT(RefSignal'LENGTH-1 downto 0);
  VARIABLE TestEvent       : VitalBoolArrayT(TestSignal'LENGTH-1 downto 0);
  VARIABLE TestDly         : TIME;
  VARIABLE RefDly          : TIME;
  VARIABLE bias            : TIME;
  VARIABLE NumTestBits     : NATURAL := TestSignal'LENGTH;
  VARIABLE NumRefBits      : NATURAL := RefSignal'LENGTH;
  VARIABLE NumChecks       : NATURAL;
```

Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE ViolationTest   : X01ArrayT(NumTestBits-1 downto 0);
    VARIABLE ViolationRef    : X01ArrayT(NumRefBits-1 downto 0);

    VARIABLE TestSignalNorm  : std_logic_vector(NumTestBits-1 downto 0)
                                := TestSignal;
    VARIABLE TestDelayNorm   : VitalDelayArraytype(NumTestBits-1 downto 0)
                                := TestDelay;
    VARIABLE RefSignalNorm   : std_logic_vector(NumRefBits-1 downto 0)
                                := RefSignal;
    VARIABLE RefDelayNorm    : VitalDelayArraytype(NumRefBits-1 downto 0)
                                := RefDelay;
    VARIABLE SetupHighNorm   : VitalDelayArraytype(SetupHigh'LENGTH-1 downto 0)
                                := SetupHigh;
    VARIABLE SetupLowNorm    : VitalDelayArraytype(SetupLow'LENGTH-1 downto 0)
                                := SetupLow;
    VARIABLE HoldHighNorm    : VitalDelayArraytype(HoldHigh'LENGTH-1 downto 0)
                                := HoldHigh;
    VARIABLE HoldLowNorm     : VitalDelayArraytype(HoldLow'LENGTH-1 downto 0)
                                := HoldLow;

    VARIABLE RefBitLow       : NATURAL;
    VARIABLE RefBitHigh      : NATURAL;
    VARIABLE EnArrayIndex    : NATURAL;
    VARIABLE TimingArrayIndex: NATURAL;
  BEGIN

    -- Initialization of working area.
    IF (TimingData.NotFirstFlag = FALSE) THEN
      TimingData.TestLastA := NEW std_logic_vector(NumTestBits-1 downto 0);
      TimingData.TestTimeA := NEW VitalTimeArrayT(NumTestBits-1 downto 0);
      TimingData.RefTimeA  := NEW VitalTimeArrayT(NumRefBits-1 downto 0);
      TimingData.RefLastA  := NEW X01ArrayT(NumRefBits-1 downto 0);
      IF (ArcType = CrossArc) THEN
        NumChecks := RefSignal'LENGTH * TestSignal'LENGTH;
      ELSE
        NumChecks := TestSignal'LENGTH;
      END IF;
      TimingData.HoldEnA   := NEW VitalBoolArrayT(NumChecks-1 downto 0);
      TimingData.SetupEnA  := NEW VitalBoolArrayT(NumChecks-1 downto 0);

      FOR i IN TestSignalNorm'RANGE LOOP
        TimingData.TestLastA(i) := To_X01(TestSignalNorm(i));
      END LOOP;

      FOR i IN RefSignalNorm'RANGE LOOP
        TimingData.RefLastA(i) := To_X01(RefSignalNorm(i));
      END LOOP;
      TimingData.NotFirstFlag := TRUE;
    END IF;

    -- Detect reference edges and record the time of the last edge
    FOR i IN RefSignalNorm'RANGE LOOP
      RefEdge(i) := EdgeSymbolMatch(TimingData.RefLastA(i),
                    To_X01(RefSignalNorm(i)), RefTransition);
      TimingData.RefLastA(i) := To_X01(RefSignalNorm(i));
      IF (RefEdge(i)) THEN
        TimingData.RefTimeA(i)  := NOW;
      END IF;
    END LOOP;

    -- Detect test (data) changes and record the time of the last change
    FOR i IN TestSignalNorm'RANGE LOOP
      TestEvent(i) := TimingData.TestLastA(i) /= To_X01Z(TestSignalNorm(i));
      TimingData.TestLastA(i) := To_X01Z(TestSignalNorm(i));
      IF (TestEvent(i)) THEN
        TimingData.TestTimeA(i) := NOW;
      END IF;
    END LOOP;

    FOR i IN ViolationTest'RANGE LOOP
      ViolationTest(i) := '0';
    END LOOP;
    FOR i IN ViolationRef'RANGE LOOP
      ViolationRef(i) := '0';
    END LOOP;

    FOR i IN TestSignalNorm'RANGE LOOP
      IF (ArcType = CrossArc) THEN
        FOR j IN RefSignalNorm'RANGE LOOP
          IF (TestEvent(i)) THEN
```

 Copyright © 2001 IEEE. All rights reserved.

```
          --TimingData.SetupEnA(i*NumRefBits+j) := TRUE;
          --IR252
          TimingData.SetupEnA(i*NumRefBits+j) := EnableSetupOnTest;
          TimingData.HoldEnA(i*NumRefBits+j)
               := TimingData.HoldEnA(i*NumRefBits+j) AND EnableHoldOnTest;
        END IF;
        IF (RefEdge(j)) THEN
          --TimingData.HoldEnA(i*NumRefBits+j) := TRUE;
          --IR252
          TimingData.HoldEnA(i*NumRefBits+j) := EnableHoldOnRef;
          TimingData.SetupEnA(i*NumRefBits+j)
               := TimingData.SetupEnA(i*NumRefBits+j) AND EnableSetupOnRef;
        END IF;
      END LOOP;
      RefBitLow := 0;
      RefBitHigh := NumRefBits-1;
      TimingArrayIndex := i;
    ELSE
      IF ArcType = SubwordArc THEN
        RefBitLow  := i / NumBitsPerSubWord;
        TimingArrayIndex := i + NumTestBits * RefBitLow;
      ELSE
        RefBitLow  := i;
        TimingArrayIndex := i;
      END IF;
      RefBitHigh := RefBitLow;
      IF TestEvent(i) THEN
        --TimingData.SetupEnA(i) := TRUE;
        --IR252
        TimingData.SetupEnA(i) := EnableSetupOnTest;
        TimingData.HoldEnA(i) := TimingData.HoldEnA(i) AND EnableHoldOnTest;
      END IF;
      IF RefEdge(RefBitLow) THEN
        --TimingData.HoldEnA(i) := TRUE;
        --IR252
        TimingData.HoldEnA(i) := EnableHoldOnRef;
        TimingData.SetupEnA(i) := TimingData.SetupEnA(i) AND EnableSetupOnRef;
      END IF;
    END IF;

    EnArrayIndex := i;
    FOR j IN RefBitLow to RefBitHigh LOOP

      IF (CheckEnabled) THEN
        TestDly := Maximum(0 ns, TestDelayNorm(i));
        RefDly  := Maximum(0 ns, RefDelayNorm(j));

        InternalTimingCheck (
          TestSignal   => TestSignalNorm(i),
          RefSignal    => RefSignalNorm(j),
          TestDelay    => TestDly,
          RefDelay     => RefDly,
          SetupHigh    => SetupHighNorm(TimingArrayIndex),
          SetupLow     => SetupLowNorm(TimingArrayIndex),
          HoldHigh     => HoldHighNorm(TimingArrayIndex),
          HoldLow      => HoldLowNorm(TimingArrayIndex),
          RefTime      => TimingData.RefTimeA(j),
          RefEdge      => RefEdge(j),
          TestTime     => TimingData.TestTimeA(i),
          TestEvent    => TestEvent(i),
          SetupEn      => TimingData.SetupEnA(EnArrayIndex),
          HoldEn       => TimingData.HoldEnA(EnArrayIndex),
          CheckInfo    => CheckInfo,
          MsgOn        => MsgOn
        );

        -- Report any detected violations and set return violation flag
        IF (CheckInfo.Violation) THEN
          IF (MsgOn) THEN
             VitalMemoryReportViolation (TestSignalName, RefSignalName, i, j,
                    TestSignal, RefSignal, HeaderMsg, CheckInfo,
                    MsgFormat, MsgSeverity );
          END IF;
          IF (XOn) THEN
              ViolationTest(i) := 'X';
              ViolationRef(j) := 'X';
          END IF;
        END IF;
      END IF;

      TimingArrayIndex := TimingArrayIndex + NumRefBits;
```

Copyright © 2001 IEEE. All rights reserved.

```
      EnArrayIndex    := EnArrayIndex + NumRefBits;

    END LOOP;
  END LOOP;

  IF (ArcType = CrossArc) THEN
    Violation := ViolationRef;
  ELSE
    IF (Violation'LENGTH = ViolationRef'LENGTH) THEN
      Violation := ViolationRef;
    ELSE
      Violation := ViolationTest;
    END IF;
  END IF;

END VitalMemorySetupHoldCheck;

-- ------------------------------------------------------------------------------
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation     : OUT    X01ArrayT;
  VARIABLE TimingData    : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal    : IN     std_logic_vector;
  CONSTANT TestSignalName: IN     STRING := "";
  CONSTANT TestDelay     : IN     VitalDelayArraytype;
  SIGNAL   RefSignal     : IN     std_logic_vector;
  CONSTANT RefSignalName : IN     STRING := "";
  CONSTANT RefDelay      : IN     VitalDelayArraytype;
  CONSTANT SetupHigh     : IN     VitalDelayArraytype;
  CONSTANT SetupLow      : IN     VitalDelayArraytype;
  CONSTANT HoldHigh      : IN     VitalDelayArraytype;
  CONSTANT HoldLow       : IN     VitalDelayArraytype;
  CONSTANT CheckEnabled  : IN     VitalBoolArrayT;
  CONSTANT RefTransition : IN     VitalEdgeSymbolType;
  CONSTANT ArcType       : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN INTEGER := 1;
  CONSTANT HeaderMsg     : IN     STRING := " ";
  CONSTANT XOn           : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat     : IN     VitalMemoryMsgFormatType;
  --IR252 3/23/98
  CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE
) IS

  VARIABLE CheckInfo       : CheckInfoType;
  VARIABLE RefEdge         : VitalBoolArrayT(RefSignal'LENGTH-1 downto 0);
  VARIABLE TestEvent       : VitalBoolArrayT(TestSignal'LENGTH-1 downto 0);
  VARIABLE TestDly         : TIME;
  VARIABLE RefDly          : TIME;
  VARIABLE bias            : TIME;
  VARIABLE NumTestBits     : NATURAL := TestSignal'LENGTH;
  VARIABLE NumRefBits      : NATURAL := RefSignal'LENGTH;
  VARIABLE NumChecks       : NATURAL;

  VARIABLE ViolationTest   : X01ArrayT(NumTestBits-1 downto 0);
  VARIABLE ViolationRef    : X01ArrayT(NumRefBits-1 downto 0);

  VARIABLE TestSignalNorm  : std_logic_vector(NumTestBits-1 downto 0)
                             := TestSignal;
  VARIABLE TestDelayNorm   : VitalDelayArraytype(NumTestBits-1 downto 0)
                             := TestDelay;
  VARIABLE RefSignalNorm   : std_logic_vector(NumRefBits-1 downto 0)
                             := RefSignal;
  VARIABLE RefDelayNorm    : VitalDelayArraytype(NumRefBits-1 downto 0)
                             := RefDelay;
  VARIABLE CheckEnNorm     : VitalBoolArrayT(NumRefBits-1 downto 0)
                             := CheckEnabled;
  VARIABLE SetupHighNorm   : VitalDelayArraytype(SetupHigh'LENGTH-1 downto 0)
                             := SetupHigh;
  VARIABLE SetupLowNorm    : VitalDelayArraytype(SetupLow'LENGTH-1 downto 0)
                             := SetupLow;
  VARIABLE HoldHighNorm    : VitalDelayArraytype(HoldHigh'LENGTH-1 downto 0)
                             := HoldHigh;
  VARIABLE HoldLowNorm     : VitalDelayArraytype(HoldLow'LENGTH-1 downto 0)
                             := HoldLow;

  VARIABLE RefBitLow       : NATURAL;
  VARIABLE RefBitHigh      : NATURAL;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE EnArrayIndex    : NATURAL;
    VARIABLE TimingArrayIndex: NATURAL;
  BEGIN

    -- Initialization of working area.
    IF (TimingData.NotFirstFlag = FALSE) THEN
      TimingData.TestLastA := NEW std_logic_vector(NumTestBits-1 downto 0);
      TimingData.TestTimeA := NEW VitalTimeArrayT(NumTestBits-1 downto 0);
      TimingData.RefTimeA  := NEW VitalTimeArrayT(NumRefBits-1 downto 0);
      TimingData.RefLastA  := NEW X01ArrayT(NumRefBits-1 downto 0);
      IF ArcType = CrossArc THEN
        NumChecks := RefSignal'LENGTH * TestSignal'LENGTH;
      ELSE
        NumChecks := TestSignal'LENGTH;
      END IF;
      TimingData.HoldEnA   := NEW VitalBoolArrayT(NumChecks-1 downto 0);
      TimingData.SetupEnA  := NEW VitalBoolArrayT(NumChecks-1 downto 0);

      FOR i IN TestSignalNorm'RANGE LOOP
        TimingData.TestLastA(i) := To_X01(TestSignalNorm(i));
      END LOOP;

      FOR i IN RefSignalNorm'RANGE LOOP
        TimingData.RefLastA(i) := To_X01(RefSignalNorm(i));
      END LOOP;
      TimingData.NotFirstFlag := TRUE;
    END IF;

    -- Detect reference edges and record the time of the last edge
    FOR i IN RefSignalNorm'RANGE LOOP
      RefEdge(i) := EdgeSymbolMatch(TimingData.RefLastA(i),
                    To_X01(RefSignalNorm(i)), RefTransition);
      TimingData.RefLastA(i) := To_X01(RefSignalNorm(i));
      IF RefEdge(i) THEN
        TimingData.RefTimeA(i)  := NOW;
      END IF;
    END LOOP;

    -- Detect test (data) changes and record the time of the last change
    FOR i IN TestSignalNorm'RANGE LOOP
      TestEvent(i) := TimingData.TestLastA(i) /= To_X01Z(TestSignalNorm(i));
      TimingData.TestLastA(i) := To_X01Z(TestSignalNorm(i));
      IF TestEvent(i) THEN
        TimingData.TestTimeA(i) := NOW;
      END IF;
    END LOOP;

    FOR i IN ViolationTest'RANGE LOOP
      ViolationTest(i) := '0';
    END LOOP;
    FOR i IN ViolationRef'RANGE LOOP
      ViolationRef(i) := '0';
    END LOOP;

    FOR i IN TestSignalNorm'RANGE LOOP
      IF (ArcType = CrossArc) THEN
        FOR j IN RefSignalNorm'RANGE LOOP
          IF (TestEvent(i)) THEN
            --TimingData.SetupEnA(i*NumRefBits+j) := TRUE;
            --IR252
            TimingData.SetupEnA(i*NumRefBits+j) := EnableSetupOnTest;
            TimingData.HoldEnA(i*NumRefBits+j)
                 := TimingData.HoldEnA(i*NumRefBits+j) AND EnableHoldOnTest;
          END IF;
          IF (RefEdge(j)) THEN
            --TimingData.HoldEnA(i*NumRefBits+j) := TRUE;
            --IR252
            TimingData.HoldEnA(i*NumRefBits+j) := EnableHoldOnRef;
            TimingData.SetupEnA(i*NumRefBits+j)
                 := TimingData.SetupEnA(i*NumRefBits+j) AND EnableSetupOnRef;
          END IF;
        END LOOP;
        RefBitLow := 0;
        RefBitHigh := NumRefBits-1;
        TimingArrayIndex := i;
      ELSE
        IF (ArcType = SubwordArc) THEN
          RefBitLow  := i / NumBitsPerSubWord;
          TimingArrayIndex := i + NumTestBits * RefBitLow;
        ELSE
          RefBitLow  := i;
```

Copyright © 2001 IEEE. All rights reserved.

```
        TimingArrayIndex := i;
      END IF;
      RefBitHigh := RefBitLow;
      IF (TestEvent(i)) THEN
        --TimingData.SetupEnA(i) := TRUE;
        --IR252
        TimingData.SetupEnA(i) := EnableSetupOnTest;
        TimingData.HoldEnA(i) := TimingData.HoldEnA(i) AND EnableHoldOnTest;
      END IF;
      IF (RefEdge(RefBitLow)) THEN
        --TimingData.HoldEnA(i) := TRUE;
        --IR252
        TimingData.HoldEnA(i) := EnableHoldOnRef;
        TimingData.SetupEnA(i) := TimingData.SetupEnA(i) AND EnableSetupOnRef;
      END IF;
    END IF;

    EnArrayIndex := i;
    FOR j IN RefBitLow to RefBitHigh LOOP
      IF (CheckEnNorm(j)) THEN
        TestDly := Maximum(0 ns, TestDelayNorm(i));
        RefDly  := Maximum(0 ns, RefDelayNorm(j));

        InternalTimingCheck (
          TestSignal   => TestSignalNorm(i),
          RefSignal    => RefSignalNorm(j),
          TestDelay    => TestDly,
          RefDelay     => RefDly,
          SetupHigh    => SetupHighNorm(TimingArrayIndex),
          SetupLow     => SetupLowNorm(TimingArrayIndex),
          HoldHigh     => HoldHighNorm(TimingArrayIndex),
          HoldLow      => HoldLowNorm(TimingArrayIndex),
          RefTime      => TimingData.RefTimeA(j),
          RefEdge      => RefEdge(j),
          TestTime     => TimingData.TestTimeA(i),
          TestEvent    => TestEvent(i),
          SetupEn      => TimingData.SetupEnA(EnArrayIndex),
          HoldEn       => TimingData.HoldEnA(EnArrayIndex),
          CheckInfo    => CheckInfo,
          MsgOn        => MsgOn
        );

        -- Report any detected violations and set return violation flag
        IF (CheckInfo.Violation) THEN
          IF (MsgOn) THEN
            VitalMemoryReportViolation (TestSignalName, RefSignalName, i, j,
                    TestSignal, RefSignal, HeaderMsg, CheckInfo,
                    MsgFormat, MsgSeverity );
          END IF;

          IF (XOn) THEN
            ViolationTest(i) := 'X';
            ViolationRef(j) := 'X';
          END IF;
        END IF;
      END IF;

      TimingArrayIndex := TimingArrayIndex + NumRefBits;
      EnArrayIndex    := EnArrayIndex + NumRefBits;
    END LOOP;
  END LOOP;

  IF (ArcType = CrossArc) THEN
    Violation := ViolationRef;
  ELSE
    IF (Violation'LENGTH = ViolationRef'LENGTH) THEN
      Violation := ViolationRef;
    ELSE
      Violation := ViolationTest;
    END IF;
  END IF;

END VitalMemorySetupHoldCheck;

-- ----------------------------------------------------------------------------
-- scalar violations not needed
-- ----------------------------------------------------------------------------
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation     : OUT    X01;
  VARIABLE TimingData    : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal    : IN     std_logic_vector;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT TestSignalName: IN     STRING := "";
    CONSTANT TestDelay     : IN     VitalDelayArraytype;
    SIGNAL   RefSignal     : IN     std_ulogic;
    CONSTANT RefSignalName : IN     STRING := "";
    CONSTANT RefDelay      : IN     TIME := 0 ns;
    CONSTANT SetupHigh     : IN     VitalDelayArraytype;
    CONSTANT SetupLow      : IN     VitalDelayArraytype;
    CONSTANT HoldHigh      : IN     VitalDelayArraytype;
    CONSTANT HoldLow       : IN     VitalDelayArraytype;
    CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
    CONSTANT RefTransition : IN     VitalEdgeSymbolType;
    CONSTANT HeaderMsg     : IN     STRING := " ";
    CONSTANT XOn           : IN     BOOLEAN := TRUE;
    CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
    CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
    CONSTANT MsgFormat     : IN     VitalMemoryMsgFormatType;
    --IR252 3/23/98
    CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;
    CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;
    CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;
    CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE
  ) IS
    VARIABLE CheckInfo     : CheckInfoType;
    VARIABLE RefEdge       : BOOLEAN;
    VARIABLE TestEvent     : VitalBoolArrayT(TestSignal'RANGE);
    VARIABLE TestDly       : TIME;
    VARIABLE RefDly        : TIME := Maximum(0 ns, RefDelay);
    VARIABLE bias          : TIME;

  BEGIN

    -- Initialization of working area.
    IF (TimingData.NotFirstFlag = FALSE) THEN
      TimingData.TestLastA := NEW std_logic_vector(TestSignal'RANGE);
      TimingData.TestTimeA := NEW VitalTimeArrayT(TestSignal'RANGE);
      TimingData.HoldEnA   := NEW VitalBoolArrayT(TestSignal'RANGE);
      TimingData.SetupEnA  := NEW VitalBoolArrayT(TestSignal'RANGE);
      FOR i IN TestSignal'RANGE LOOP
        TimingData.TestLastA(i) := To_X01(TestSignal(i));
      END LOOP;
      TimingData.RefLast := To_X01(RefSignal);
      TimingData.NotFirstFlag := TRUE;
    END IF;

    -- Detect reference edges and record the time of the last edge
    RefEdge := EdgeSymbolMatch(TimingData.RefLast, To_X01(RefSignal),
                   RefTransition);
    TimingData.RefLast := To_X01(RefSignal);
    IF (RefEdge) THEN
      TimingData.RefTime     := NOW;
      --TimingData.HoldEnA.all := (TestSignal'RANGE=>TRUE);
      --IR252 3/23/98
      FOR i IN TestSignal'RANGE LOOP
        TimingData.SetupEnA(i)
          := TimingData.SetupEnA(i) AND EnableSetupOnRef;
        TimingData.HoldEnA(i)  := EnableHoldOnRef;
      END LOOP;
    END IF;

    -- Detect test (data) changes and record the time of the last change
    FOR i IN TestSignal'RANGE LOOP
      TestEvent(i) := TimingData.TestLastA(i) /= To_X01Z(TestSignal(i));
      TimingData.TestLastA(i) := To_X01Z(TestSignal(i));
      IF TestEvent(i) THEN
        TimingData.SetupEnA(i)  := EnableSetupOnTest ; --IR252 3/23/98
        TimingData.HoldEnA(i) := TimingData.HoldEnA(i) AND EnableHoldOnTest ;
                                                --IR252 3/23/98
        TimingData.TestTimeA(i) := NOW;
        --TimingData.SetupEnA(i)  := TRUE;
        TimingData.TestTime := NOW;
      END IF;
    END LOOP;

    Violation := '0';
    FOR i IN TestSignal'RANGE LOOP
      IF (CheckEnabled) THEN
        TestDly := Maximum(0 ns, TestDelay(i));
        InternalTimingCheck (
          TestSignal   => TestSignal(i),
          RefSignal    => RefSignal,
          TestDelay    => TestDly,
```

Copyright © 2001 IEEE. All rights reserved.

```
            RefDelay     => RefDly,
            SetupHigh    => SetupHigh(i),
            SetupLow     => SetupLow(i),
            HoldHigh     => HoldHigh(i),
            HoldLow      => HoldLow(i),
            RefTime      => TimingData.RefTime,
            RefEdge      => RefEdge,
            TestTime     => TimingData.TestTimeA(i),
            TestEvent    => TestEvent(i),
            SetupEn      => TimingData.SetupEnA(i),
            HoldEn       => TimingData.HoldEnA(i),
            CheckInfo    => CheckInfo,
            MsgOn        => MsgOn
          );

          -- Report any detected violations and set return violation flag
          IF CheckInfo.Violation THEN
            IF (MsgOn) THEN
              VitalMemoryReportViolation (TestSignalName, RefSignalName, i ,
                      HeaderMsg, CheckInfo, MsgFormat, MsgSeverity );
            END IF;
            IF (XOn) THEN
              Violation := 'X';
            END IF;
          END IF;
        END IF;
      END LOOP;

    END VitalMemorySetupHoldCheck;

    -- ------------------------------------------------------------------------------
    PROCEDURE VitalMemorySetupHoldCheck (
      VARIABLE Violation     : OUT    X01;
      VARIABLE TimingData    : INOUT  VitalMemoryTimingDataType;
      SIGNAL   TestSignal    : IN     std_logic_vector;
      CONSTANT TestSignalName: IN     STRING := "";
      CONSTANT TestDelay     : IN     VitalDelayArraytype;
      SIGNAL   RefSignal     : IN     std_logic_vector;
      CONSTANT RefSignalName : IN     STRING := "";
      CONSTANT RefDelay      : IN     VitalDelayArraytype;
      CONSTANT SetupHigh     : IN     VitalDelayArraytype;
      CONSTANT SetupLow      : IN     VitalDelayArraytype;
      CONSTANT HoldHigh      : IN     VitalDelayArraytype;
      CONSTANT HoldLow       : IN     VitalDelayArraytype;
      CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
      CONSTANT RefTransition : IN     VitalEdgeSymbolType;
      CONSTANT HeaderMsg     : IN     STRING := " ";
      CONSTANT XOn           : IN     BOOLEAN := TRUE;
      CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
      CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
      CONSTANT ArcType       : IN     VitalMemoryArcType := CrossArc;
      CONSTANT NumBitsPerSubWord : IN INTEGER := 1;
      CONSTANT MsgFormat     : IN     VitalMemoryMsgFormatType;
      --IR252 3/23/98
      CONSTANT EnableSetupOnTest : IN   BOOLEAN := TRUE;
      CONSTANT EnableSetupOnRef  : IN   BOOLEAN := TRUE;
      CONSTANT EnableHoldOnRef   : IN   BOOLEAN := TRUE;
      CONSTANT EnableHoldOnTest  : IN   BOOLEAN := TRUE
    ) IS
      VARIABLE CheckInfo       : CheckInfoType;
      VARIABLE RefEdge         : VitalBoolArrayT(RefSignal'LENGTH-1 downto 0);
      VARIABLE TestEvent       : VitalBoolArrayT(TestSignal'LENGTH-1 downto 0);
      VARIABLE TestDly         : TIME;
      VARIABLE RefDly          : TIME;
      VARIABLE bias            : TIME;
      VARIABLE NumTestBits     : NATURAL := TestSignal'LENGTH;
      VARIABLE NumRefBits      : NATURAL := RefSignal'LENGTH;
      VARIABLE NumChecks       : NATURAL;

      VARIABLE TestSignalNorm  : std_logic_vector(NumTestBits-1 downto 0)
                                  := TestSignal;
      VARIABLE TestDelayNorm   : VitalDelayArraytype(NumTestBits-1 downto 0)
                                  := TestDelay;
      VARIABLE RefSignalNorm   : std_logic_vector(NumRefBits-1 downto 0)
                                  := RefSignal;
      VARIABLE RefDelayNorm    : VitalDelayArraytype(NumRefBits-1 downto 0)
                                  := RefDelay;
      VARIABLE SetupHighNorm   : VitalDelayArraytype(SetupHigh'LENGTH-1 downto 0)
                                  := SetupHigh;
      VARIABLE SetupLowNorm    : VitalDelayArraytype(SetupLow'LENGTH-1 downto 0)
                                  := SetupLow;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE HoldHighNorm    : VitalDelayArraytype(HoldHigh'LENGTH-1 downto 0)
                                 := HoldHigh;
    VARIABLE HoldLowNorm     : VitalDelayArraytype(HoldLow'LENGTH-1 downto 0)
                                 := HoldLow;

    VARIABLE RefBitLow       : NATURAL;
    VARIABLE RefBitHigh      : NATURAL;
    VARIABLE EnArrayIndex    : NATURAL;
    VARIABLE TimingArrayIndex: NATURAL;
  BEGIN

    -- Initialization of working area.
    IF (TimingData.NotFirstFlag = FALSE) THEN
      TimingData.TestLastA := NEW std_logic_vector(NumTestBits-1 downto 0);
      TimingData.TestTimeA := NEW VitalTimeArrayT(NumTestBits-1 downto 0);
      TimingData.RefTimeA  := NEW VitalTimeArrayT(NumRefBits-1 downto 0);
      TimingData.RefLastA  := NEW X01ArrayT(NumRefBits-1 downto 0);
      IF (ArcType = CrossArc) THEN
        NumChecks := RefSignal'LENGTH * TestSignal'LENGTH;
      ELSE
        NumChecks := TestSignal'LENGTH;
      END IF;
      TimingData.HoldEnA   := NEW VitalBoolArrayT(NumChecks-1 downto 0);
      TimingData.SetupEnA  := NEW VitalBoolArrayT(NumChecks-1 downto 0);

      FOR i IN TestSignalNorm'RANGE LOOP
        TimingData.TestLastA(i) := To_X01(TestSignalNorm(i));
      END LOOP;

      FOR i IN RefSignalNorm'RANGE LOOP
        TimingData.RefLastA(i) := To_X01(RefSignalNorm(i));
      END LOOP;
      TimingData.NotFirstFlag := TRUE;
    END IF;

    -- Detect reference edges and record the time of the last edge
    FOR i IN RefSignalNorm'RANGE LOOP
      RefEdge(i) := EdgeSymbolMatch(TimingData.RefLastA(i),
                      To_X01(RefSignalNorm(i)), RefTransition);
      TimingData.RefLastA(i) := To_X01(RefSignalNorm(i));
      IF (RefEdge(i)) THEN
        TimingData.RefTimeA(i)  := NOW;
      END IF;
    END LOOP;

    -- Detect test (data) changes and record the time of the last change
    FOR i IN TestSignalNorm'RANGE LOOP
      TestEvent(i) := TimingData.TestLastA(i) /= To_X01Z(TestSignalNorm(i));
      TimingData.TestLastA(i) := To_X01Z(TestSignalNorm(i));
      IF (TestEvent(i)) THEN
        TimingData.TestTimeA(i) := NOW;
      END IF;
    END LOOP;

    FOR i IN TestSignalNorm'RANGE LOOP
      IF (ArcType = CrossArc) THEN
        FOR j IN RefSignalNorm'RANGE LOOP
          IF (TestEvent(i)) THEN
            --TimingData.SetupEnA(i*NumRefBits+j) := TRUE;
            --IR252
            TimingData.SetupEnA(i*NumRefBits+j) := EnableSetupOnTest;
            TimingData.HoldEnA(i*NumRefBits+j)
                   := TimingData.HoldEnA(i*NumRefBits+j) AND EnableHoldOnTest;
          END IF;
          IF (RefEdge(j)) THEN
            --TimingData.HoldEnA(i*NumRefBits+j) := TRUE;
            --IR252
            TimingData.HoldEnA(i*NumRefBits+j) := EnableHoldOnRef;
            TimingData.SetupEnA(i*NumRefBits+j)
                   := TimingData.SetupEnA(i*NumRefBits+j) AND EnableSetupOnRef;
          END IF;
        END LOOP;
        RefBitLow := 0;
        RefBitHigh := NumRefBits-1;
        TimingArrayIndex := i;
      ELSE
        IF (ArcType = SubwordArc) THEN
          RefBitLow  := i / NumBitsPerSubWord;
          TimingArrayIndex := i + NumTestBits * RefBitLow;
        ELSE
          RefBitLow  := i;
```

Copyright © 2001 IEEE. All rights reserved.

```
        TimingArrayIndex := i;
      END IF;
      RefBitHigh := RefBitLow;
      IF (TestEvent(i)) THEN
        --TimingData.SetupEnA(i) := TRUE;
        --IR252
        TimingData.SetupEnA(i) := EnableSetupOnTest;
        TimingData.HoldEnA(i) := TimingData.HoldEnA(i) AND EnableHoldOnTest;
      END IF;
      IF (RefEdge(RefBitLow)) THEN
        --TimingData.HoldEnA(i) := TRUE;
        --IR252
        TimingData.HoldEnA(i) := EnableHoldOnRef;
        TimingData.SetupEnA(i) := TimingData.SetupEnA(i) AND EnableSetupOnRef;
      END IF;
    END IF;

    EnArrayIndex := i;
    Violation := '0';
    FOR j IN RefBitLow to RefBitHigh LOOP

      IF (CheckEnabled) THEN
        TestDly := Maximum(0 ns, TestDelayNorm(i));
        RefDly  := Maximum(0 ns, RefDelayNorm(j));

        InternalTimingCheck (
          TestSignal   => TestSignalNorm(i),
          RefSignal    => RefSignalNorm(j),
          TestDelay    => TestDly,
          RefDelay     => RefDly,
          SetupHigh    => SetupHighNorm(TimingArrayIndex),
          SetupLow     => SetupLowNorm(TimingArrayIndex),
          HoldHigh     => HoldHighNorm(TimingArrayIndex),
          HoldLow      => HoldLowNorm(TimingArrayIndex),
          RefTime      => TimingData.RefTimeA(j),
          RefEdge      => RefEdge(j),
          TestTime     => TimingData.TestTimeA(i),
          TestEvent    => TestEvent(i),
          SetupEn      => TimingData.SetupEnA(EnArrayIndex),
          HoldEn       => TimingData.HoldEnA(EnArrayIndex),
          CheckInfo    => CheckInfo,
          MsgOn        => MsgOn
        );

        -- Report any detected violations and set return violation flag
        IF (CheckInfo.Violation) THEN
          IF (MsgOn) THEN
            VitalMemoryReportViolation (TestSignalName, RefSignalName, i, j,
                    TestSignal, RefSignal, HeaderMsg, CheckInfo,
                    MsgFormat, MsgSeverity );
          END IF;

          IF (XOn) THEN
            Violation := 'X';
          END IF;
        END IF;
      END IF;

      TimingArrayIndex := TimingArrayIndex + NumRefBits;
      EnArrayIndex    := EnArrayIndex + NumRefBits;

    END LOOP;
  END LOOP;

END VitalMemorySetupHoldCheck;

-- ----------------------------------------------------------------------------
PROCEDURE  VitalMemoryPeriodPulseCheck  (
  VARIABLE Violation      : OUT   X01;
  VARIABLE PeriodData     : INOUT VitalPeriodDataArrayType;
  SIGNAL   TestSignal     : IN    std_logic_vector;
  CONSTANT TestSignalName : IN    STRING := "";
  CONSTANT TestDelay      : IN    VitalDelayArraytype;
  CONSTANT Period         : IN    VitalDelayArraytype;
  CONSTANT PulseWidthHigh : IN    VitalDelayArraytype;
  CONSTANT PulseWidthLow  : IN    VitalDelayArraytype;
  CONSTANT CheckEnabled   : IN    BOOLEAN := TRUE;
  CONSTANT HeaderMsg      : IN    STRING := " ";
  CONSTANT XOn            : IN    BOOLEAN := TRUE;
  CONSTANT MsgOn          : IN    BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN    SEVERITY_LEVEL := WARNING;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT MsgFormat      : IN    VitalMemoryMsgFormatType
  ) IS
    VARIABLE TestDly   : VitalDelayType;
    VARIABLE CheckInfo : CheckInfoType;
    VARIABLE PeriodObs : VitalDelayType;
    VARIABLE PulseTest : BOOLEAN;
    VARIABLE PeriodTest: BOOLEAN;
    VARIABLE TestValue : X01;
  BEGIN

    FOR i IN TestSignal'RANGE LOOP
      TestDly := Maximum(0 ns, TestDelay(i));
      TestValue := To_X01(TestSignal(i));

      IF (PeriodData(i).NotFirstFlag = FALSE) THEN
        PeriodData(i).Rise := -Maximum(Period(i),
            Maximum(PulseWidthHigh(i),PulseWidthLow(i)));
        PeriodData(i).Fall := -Maximum(Period(i),
            Maximum(PulseWidthHigh(i),PulseWidthLow(i)));
        PeriodData(i).Last := TestValue;
        PeriodData(i).NotFirstFlag := TRUE;
      END IF;

      -- Initialize for no violation
      Violation := '0';

      -- No violation possible if no test signal change
      NEXT WHEN (PeriodData(i).Last = TestValue);

      -- record starting pulse times
      IF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'P')) THEN
        -- Compute period times, then record the High Rise Time
        PeriodObs := NOW - PeriodData(i).Rise;
        PeriodData(i).Rise := NOW;
        PeriodTest := TRUE;
      ELSIF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'N')) THEN
        -- Compute period times, then record the Low Fall Time
        PeriodObs := NOW - PeriodData(i).Fall;
        PeriodData(i).Fall := NOW;
        PeriodTest := TRUE;
      ELSE
        PeriodTest := FALSE;
      END IF;

      -- do checks on pulse ends
      IF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'p')) THEN
        -- Compute pulse times
        CheckInfo.ObsTime := NOW - PeriodData(i).Fall;
        CheckInfo.ExpTime := PulseWidthLow(i);
        PulseTest := TRUE;
      ELSIF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'n')) THEN
        -- Compute pulse times
        CheckInfo.ObsTime := NOW - PeriodData(i).Rise;
        CheckInfo.ExpTime := PulseWidthHigh(i);
        PulseTest := TRUE;
      ELSE
        PulseTest := FALSE;
      END IF;

      IF (PulseTest AND CheckEnabled) THEN
        -- Verify Pulse Width [ignore 1st edge]
        IF (CheckInfo.ObsTime < CheckInfo.ExpTime) THEN
          IF (XOn) THEN
           Violation := 'X';
          END IF;
          IF (MsgOn) THEN
            CheckInfo.Violation := TRUE;
            CheckInfo.CheckKind := PulseWidCheck;
            CheckInfo.DetTime   := NOW - TestDly;
            CheckInfo.State     := PeriodData(i).Last;
            VitalMemoryReportViolation (TestSignalName, "", i,
                 HeaderMsg, CheckInfo, MsgFormat, MsgSeverity );
          END IF; -- MsgOn
        END IF;
      END IF;

      IF (PeriodTest AND CheckEnabled) THEN
        -- Verify the Period [ignore 1st edge]
        CheckInfo.ObsTime := PeriodObs;
        CheckInfo.ExpTime := Period(i);
        IF ( CheckInfo.ObsTime < CheckInfo.ExpTime ) THEN
```

Copyright © 2001 IEEE. All rights reserved.

```
          IF (XOn) THEN
            Violation := 'X';
          END IF;
          IF (MsgOn) THEN
            CheckInfo.Violation := TRUE;
            CheckInfo.CheckKind := PeriodCheck;
            CheckInfo.DetTime   := NOW - TestDly;
            CheckInfo.State     := TestValue;
            VitalMemoryReportViolation (TestSignalName, "", i,
                 HeaderMsg, CheckInfo, MsgFormat, MsgSeverity );
          END IF; -- MsgOn
        END IF;
      END IF;

      PeriodData(i).Last := TestValue;
    END LOOP;

  END VitalMemoryPeriodPulseCheck;

  -- ----------------------------------------------------------------------------
  PROCEDURE  VitalMemoryPeriodPulseCheck  (
    VARIABLE Violation      : OUT   X01ArrayT;
    VARIABLE PeriodData     : INOUT VitalPeriodDataArrayType;
    SIGNAL   TestSignal     : IN    std_logic_vector;
    CONSTANT TestSignalName : IN    STRING := "";
    CONSTANT TestDelay      : IN    VitalDelayArraytype;
    CONSTANT Period         : IN    VitalDelayArraytype;
    CONSTANT PulseWidthHigh : IN    VitalDelayArraytype;
    CONSTANT PulseWidthLow  : IN    VitalDelayArraytype;
    CONSTANT CheckEnabled   : IN    BOOLEAN := TRUE;
    CONSTANT HeaderMsg      : IN    STRING := " ";
    CONSTANT XOn            : IN    BOOLEAN := TRUE;
    CONSTANT MsgOn          : IN    BOOLEAN := TRUE;
    CONSTANT MsgSeverity    : IN    SEVERITY_LEVEL := WARNING;
    CONSTANT MsgFormat      : IN    VitalMemoryMsgFormatType
  )IS
    VARIABLE TestDly   : VitalDelayType;
    VARIABLE CheckInfo : CheckInfoType;
    VARIABLE PeriodObs : VitalDelayType;
    VARIABLE PulseTest : BOOLEAN;
    VARIABLE PeriodTest: BOOLEAN;
    VARIABLE TestValue : X01;
  BEGIN

    FOR i IN TestSignal'RANGE LOOP
      TestDly := Maximum(0 ns, TestDelay(i));
      TestValue := To_X01(TestSignal(i));

      IF (PeriodData(i).NotFirstFlag = FALSE) THEN
        PeriodData(i).Rise := -Maximum(Period(i),
            Maximum(PulseWidthHigh(i),PulseWidthLow(i)));
        PeriodData(i).Fall := -Maximum(Period(i),
            Maximum(PulseWidthHigh(i),PulseWidthLow(i)));
        PeriodData(i).Last := TestValue;
        PeriodData(i).NotFirstFlag := TRUE;
      END IF;

      -- Initialize for no violation
      Violation(i) := '0';

      -- No violation possible if no test signal change
      NEXT WHEN (PeriodData(i).Last = TestValue);

      -- record starting pulse times
      IF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'P')) THEN
        -- Compute period times, then record the High Rise Time
        PeriodObs := NOW - PeriodData(i).Rise;
        PeriodData(i).Rise := NOW;
        PeriodTest := TRUE;
      ELSIF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'N')) THEN
        -- Compute period times, then record the Low Fall Time
        PeriodObs := NOW - PeriodData(i).Fall;
        PeriodData(i).Fall := NOW;
        PeriodTest := TRUE;
      ELSE
        PeriodTest := FALSE;
      END IF;

      -- do checks on pulse ends
      IF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'p')) THEN
        -- Compute pulse times
```

 Copyright © 2001 IEEE. All rights reserved.

```
        CheckInfo.ObsTime := NOW - PeriodData(i).Fall;
        CheckInfo.ExpTime := PulseWidthLow(i);
        PulseTest := TRUE;
      ELSIF (EdgeSymbolMatch(PeriodData(i).Last, TestValue, 'n')) THEN
        -- Compute pulse times
        CheckInfo.ObsTime := NOW - PeriodData(i).Rise;
        CheckInfo.ExpTime := PulseWidthHigh(i);
        PulseTest := TRUE;
      ELSE
        PulseTest := FALSE;
      END IF;

      IF (PulseTest AND CheckEnabled) THEN
        -- Verify Pulse Width [ignore 1st edge]
        IF (CheckInfo.ObsTime < CheckInfo.ExpTime) THEN
          IF (XOn) THEN
            Violation(i) := 'X';
          END IF;
          IF (MsgOn) THEN
            CheckInfo.Violation := TRUE;
            CheckInfo.CheckKind := PulseWidCheck;
            CheckInfo.DetTime   := NOW - TestDly;
            CheckInfo.State     := PeriodData(i).Last;
            VitalMemoryReportViolation (TestSignalName, "", i,
                HeaderMsg, CheckInfo, MsgFormat, MsgSeverity );
          END IF; -- MsgOn
        END IF;
      END IF;

      IF (PeriodTest AND CheckEnabled) THEN
        -- Verify the Period [ignore 1st edge]
        CheckInfo.ObsTime := PeriodObs;
        CheckInfo.ExpTime := Period(i);
        IF ( CheckInfo.ObsTime < CheckInfo.ExpTime ) THEN
          IF (XOn) THEN
            Violation(i) := 'X';
          END IF;
          IF (MsgOn) THEN
            CheckInfo.Violation := TRUE;
            CheckInfo.CheckKind := PeriodCheck;
            CheckInfo.DetTime   := NOW - TestDly;
            CheckInfo.State     := TestValue;
            VitalMemoryReportViolation (TestSignalName, "", i,
                HeaderMsg, CheckInfo, MsgFOrmat, MsgSeverity );
          END IF; -- MsgOn
        END IF;
      END IF;

      PeriodData(i).Last := TestValue;
    END LOOP;

  END VitalMemoryPeriodPulseCheck;

  -- ----------------------------------------------------------------------------
  -- Functionality Section
  -- ----------------------------------------------------------------------------

  -- Look-up table.  Given an int, we can get the 4-bit bit_vector.
  TYPE HexToBitvTableType IS  ARRAY (NATURAL RANGE <>) OF
    std_logic_vector(3 DOWNTO 0) ;

  CONSTANT HexToBitvTable : HexToBitvTableType (0 TO 15) :=
    (
      "0000", "0001", "0010", "0011",
      "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011",
      "1100", "1101", "1110", "1111"
    ) ;

  -- ----------------------------------------------------------------------------
  -- Misc Utilities Local Utilities
  -- ----------------------------------------------------------------------------

  -- ----------------------------------------------------------------------------
  -- Procedure:   IsSpace
  -- Parameters:  ch      -- input character
  -- Description: Returns TRUE or FALSE depending on the input character
  --              being white space or not.
  -- ----------------------------------------------------------------------------
  FUNCTION IsSpace (ch : character)
  RETURN boolean IS
```

Copyright © 2001 IEEE. All rights reserved.

```
BEGIN
  RETURN ((ch = ‘ ‘) OR (ch = CR) OR (ch = HT) OR (ch = NUL));
END IsSpace;

-- ----------------------------------------------------------------------------
-- Procedure:   LenOfString
-- Parameters:  Str         -- input string
-- Description: Returns the NATURAL length of the input string.
--              as terminated by the first NUL character.
-- ----------------------------------------------------------------------------
FUNCTION LenOfString (Str : STRING)
RETURN NATURAL IS
  VARIABLE StrRight : NATURAL;
BEGIN
  StrRight := Str’RIGHT;
  FOR i IN Str’RANGE LOOP
    IF (Str(i) = NUL) THEN
      StrRight := i - 1;
      EXIT;
    END IF;
  END LOOP;
  RETURN (StrRight);
END LenOfString;

-- ----------------------------------------------------------------------------
-- Procedure:   HexToInt
-- Parameters:  Hex         -- input character or string
-- Description: Converts input character or string interpreted as a
--              hexadecimal representation to integer value.
-- ----------------------------------------------------------------------------
FUNCTION HexToInt(Hex : CHARACTER) RETURN INTEGER IS
  CONSTANT HexChars : STRING := “0123456789ABCDEFabcdef”;
  CONSTANT XHiChar  : CHARACTER := ‘X’;
  CONSTANT XLoChar  : CHARACTER := ‘x’;
BEGIN
  IF (Hex = XLoChar OR Hex = XHiChar) THEN
    RETURN (23);
  END IF;
  FOR i IN 1 TO 16 LOOP
    IF(Hex = HexChars(i)) THEN
      RETURN (i-1);
    END IF;
  END LOOP;
  FOR i IN 17 TO 22 LOOP
    IF (Hex = HexChars(i)) THEN
      RETURN (i-7);
    END IF;
  END LOOP;
  ASSERT FALSE REPORT
    “Invalid character received by HexToInt function”
    SEVERITY WARNING;
  RETURN (0);
END HexToInt;

-- ----------------------------------------------------------------------------
FUNCTION HexToInt (Hex : STRING) RETURN INTEGER IS
  VARIABLE Value  : INTEGER := 0;
  VARIABLE Length : INTEGER;
BEGIN
  Length := LenOfString(hex);
  IF (Length > 8) THEN
    ASSERT FALSE REPORT
      “Invalid string length received by HexToInt function”
      SEVERITY WARNING;
  ELSE
    FOR i IN 1 TO Length LOOP
      Value := Value + HexToInt(Hex(i)) * 16 ** (Length - i);
    END LOOP;
  END IF;
  RETURN (Value);
END HexToInt;

-- ----------------------------------------------------------------------------
-- Procedure:   HexToBitv
-- Parameters:  Hex         -- Input hex string
-- Description: Converts input hex string to a std_logic_vector
-- ----------------------------------------------------------------------------
FUNCTION HexToBitv(
  Hex     : STRING
) RETURN std_logic_vector is
  VARIABLE   Index       : INTEGER := 0 ;
```

 Copyright © 2001 IEEE. All rights reserved.

```
  VARIABLE   ValHexToInt : INTEGER ;
  VARIABLE   BitsPerHex  : INTEGER := 4 ; -- Denotes no. of bits per hex char.
  VARIABLE   HexLen      : NATURAL := (BitsPerHex * LenOfString(Hex)) ;
  VARIABLE   TableVal    : std_logic_vector(3 DOWNTO 0) ;
  VARIABLE   Result      : std_logic_vector(HexLen-1 DOWNTO 0) ;
BEGIN
  --  Assign 4-bit wide bit vector to result directly from a look-up table.
  Index := 0 ;
  WHILE ( Index < HexLen )  LOOP
    ValHexToInt := HexToInt( Hex((HexLen - Index)/BitsPerHex ) );
    IF ( ValHexToInt = 23 ) THEN
      TableVal := "XXXX";
    ELSE
      -- Look up from the table.
      TableVal := HexToBitvTable( ValHexToInt ) ;
    END IF;
    -- Assign now.
    Result(Index+3 DOWNTO Index) := TableVal ;
    -- Get ready for next block of 4-bits.
    Index := Index + 4 ;
  END LOOP ;
  RETURN Result ;
END HexToBitv ;

-- ---------------------------------------------------------------------------
-- Procedure:   BinToBitv
-- Parameters:  Bin         -- Input bin string
-- Description: Converts input bin string to a std_logic_vector
-- ---------------------------------------------------------------------------
FUNCTION BinToBitv(
  Bin     : STRING
) RETURN std_logic_vector is
  VARIABLE   Index        : INTEGER := 0 ;
  VARIABLE   Length       : NATURAL := LenOfString(Bin);
  VARIABLE   BitVal       : std_ulogic;
  VARIABLE   Result       : std_logic_vector(Length-1 DOWNTO 0) ;
BEGIN
  Index := 0 ;
  WHILE ( Index < Length )  LOOP
    IF (Bin(Length-Index) = '0' ) THEN
      BitVal := '0';
    ELSIF (Bin(Length-Index) = '1' ) THEN
      BitVal := '1';
    ELSE
      BitVal := 'X';
    END IF ;
    -- Assign now.
    Result(Index) := BitVal ;
    Index := Index + 1 ;
  END LOOP ;
  RETURN Result ;
END BinToBitv ;

-- ---------------------------------------------------------------------------
-- For Memory Table Modeling
-- ---------------------------------------------------------------------------

TYPE To_MemoryCharType IS ARRAY (VitalMemorySymbolType) OF CHARACTER;
CONSTANT To_MemoryChar : To_MemoryCharType :=
  ( '/' , '\' , 'P' , 'N' , 'r' , 'f' , 'p' , 'n' , 'R' , 'F' , '^' , 'v' ,
    'E' , 'A' , 'D' , '*' , 'X' , '0' , '1' , '-' , 'B' , 'Z' , 'S' ,
    'g' , 'u' , 'i' , 'G' , 'U' , 'I' ,
    'w' , 's' ,
    'c' , 'l' , 'd' , 'e' , 'C' , 'L' ,
    'M' , 'm' , 't' );

TYPE ValidMemoryTableInputType IS ARRAY (VitalMemorySymbolType) OF BOOLEAN;
CONSTANT ValidMemoryTableInput : ValidMemoryTableInputType :=
    -- '/' ,    '\' ,    'P' ,    'N' ,    'r' ,    'f' ,
  (   TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,
    -- 'p' ,    'n' ,    'R' ,    'F' ,    '^' ,    'v' ,
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,
    -- 'E' ,    'A' ,    'D' ,    '*' ,
      TRUE,  TRUE,  TRUE,  TRUE,
    -- 'X' ,    '0' ,    '1' ,    '-' ,    'B' ,    'Z' ,
      TRUE,  TRUE,  TRUE,  TRUE,  TRUE, FALSE,
    -- 'S' ,
      TRUE,
    -- 'g' ,    'u' ,    'i' ,    'G' ,    'U' ,    'I' ,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
    -- 'w' ,    's' ,
```

Copyright © 2001 IEEE. All rights reserved.

```
        FALSE, FALSE,
      -- 'c',    'l',    'd',    'e',    'C',    'L',
        FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
    -- 'M',    'm',    't'
        FALSE, FALSE, FALSE);

  TYPE MemoryTableMatchType IS ARRAY (X01,X01,VitalMemorySymbolType) OF BOOLEAN;
  -- last value, present value, table symbol
  CONSTANT MemoryTableMatch : MemoryTableMatchType :=  (
    ( -- X (lastvalue)
      -- /     \     P     N     r     f
      -- p     n     R     F     ^     v
      -- E     A     D     *
      -- X     0     1     -     B     Z     S
      -- g     u     i     G     U     I
      -- w     s
      -- c     l     d     e,    C     L
      -- m     t
      ( FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,
        TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE),
      ( FALSE,FALSE,FALSE,TRUE, FALSE,FALSE,
        FALSE,FALSE,FALSE,TRUE, FALSE,TRUE,
        TRUE, FALSE,TRUE, TRUE,
        FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE),
      ( FALSE,FALSE,TRUE, FALSE,FALSE,FALSE,
        FALSE,FALSE,TRUE, FALSE,TRUE, FALSE,
        TRUE, TRUE, FALSE,TRUE,
        FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE)
      ),

      (-- 0 (lastvalue)
      -- /     \     P     N     r     f
      -- p     n     R     F     ^     v
      -- E     A     D     *
      -- X     0     1     -     B     Z     S
      -- g     u     i     G     U     I
      -- w     s
      -- c     l     d     e,    C     L
      -- m     t
      ( FALSE,FALSE,FALSE,FALSE,TRUE, FALSE,
        TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
        FALSE,TRUE, FALSE,TRUE,
        TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE),
      ( FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,
        FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,TRUE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE),
      ( TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
        TRUE, FALSE,TRUE, FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,TRUE,
        FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE)
      ),

      (-- 1 (lastvalue)
      -- /     \     P     N     r     f
```

 Copyright © 2001 IEEE. All rights reserved.

```
    -- p     n     R     F     ^     v
    -- E     A     D     *
    -- X     0     1     -     B     Z     S
    -- g     u     i     G     U     I
    -- w     s
    -- c     l     d     e,    C     L
    -- m     t
    ( FALSE,FALSE,FALSE,FALSE,FALSE,TRUE ,
      FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
      FALSE,FALSE,TRUE, TRUE,
      TRUE, FALSE,FALSE,TRUE, FALSE,FALSE,FALSE,
      FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,
      FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,FALSE),
    ( FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
      FALSE,TRUE, FALSE,TRUE, FALSE,FALSE,
      FALSE,FALSE,FALSE,TRUE,
      FALSE,TRUE, FALSE,TRUE, TRUE, FALSE,FALSE,
      FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,
      FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,FALSE),
    ( FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,TRUE, TRUE, TRUE, FALSE,TRUE,
      FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,
      FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
      FALSE,FALSE,FALSE)
    )
  );


-- ----------------------------------------------------------------------------
-- Error Message Types and Tables
-- ----------------------------------------------------------------------------

TYPE VitalMemoryErrorType IS (
  ErrGoodAddr,   -- 'g' Good address (no transition)
  ErrUnknAddr,   -- 'u' 'X' levels in address (no transition)
  ErrInvaAddr,   -- 'i' Invalid address (no transition)
  ErrGoodTrAddr, -- 'G' Good address (with transition)
  ErrUnknTrAddr, -- 'U' 'X' levels in address (with transition)
  ErrInvaTrAddr, -- 'I' Invalid address (with transition)
  ErrWrDatMem,   -- 'w' Writing data to memory
  ErrNoChgMem,   -- 's' Retaining previous memory contents
  ErrCrAllMem,   -- 'c' Corrupting entire memory with 'X'
  ErrCrWrdMem,   -- 'l' Corrupting a word in memory with 'X'
  ErrCrBitMem,   -- 'd' Corrupting a single bit in memory with 'X'
  ErrCrDatMem,   -- 'e' Corrupting a word with 'X' based on data in
  ErrCrAllSubMem,-- 'C' Corrupting a sub-word entire memory with 'X'
  ErrCrWrdSubMem,-- 'L' Corrupting a sub-word in memory with 'X'
  ErrCrBitSubMem,-- 'D' Corrupting a single bit of a memory sub-word with 'X'
  ErrCrDatSubMem,-- 'E' Corrupting a sub-word with 'X' based on data in
  ErrCrWrdOut,   -- 'l' Corrupting data out with 'X'
  ErrCrBitOut,   -- 'd' Corrupting a single bit of data out with 'X'
  ErrCrDatOut,   -- 'e' Corrupting data out with 'X' based on data in
  ErrCrWrdSubOut,-- 'L' Corrupting data out sub-word with 'X'
  ErrCrBitSubOut,-- 'D' Corrupting a single bit of data out sub-word with 'X'
  ErrCrDatSubOut,-- 'E' Corrupting data out sub-word with 'X' based on data in
  ErrImplOut,    -- 'M' Implicit read from memory to data out
  ErrReadOut,    -- 'm' Reading data from memory to data out
  ErrAssgOut,    -- 't' Transfering from data in to data out
  ErrAsgXOut,    -- 'X' Assigning unknown level to data out
  ErrAsg0Out,    -- '0' Assigning low level to data out
  ErrAsg1Out,    -- '1' Assigning high level to data out
  ErrAsgZOut,    -- 'Z' Assigning high impedence to data out
  ErrAsgSOut,    -- 'S' Keeping data out at steady value
  ErrAsgXMem,    -- 'X' Assigning unknown level to memory location
  ErrAsg0Mem,    -- '0' Assigning low level to memory location
  ErrAsg1Mem,    -- '1' Assigning high level to memory location
  ErrAsgZMem,    -- 'Z' Assigning high impedence to memory location
  ErrDefMemAct,  --     No memory table match, using default action
  ErrInitMem,    --     Initialize memory contents
  ErrMcpWrCont,  --     Memory cross port to same port write contention
  ErrMcpCpCont,  --     Memory cross port read/write data/memory contention
  ErrMcpCpRead,  --     Memory cross port read to same port
  ErrMcpRdWrCo,  --     Memory cross port read/write data only contention
  ErrMcpCpWrCont,--     Memory cross port to cross port write contention
```

```
  ErrUnknMemDo,  --     Unknown memory action
  ErrUnknDatDo,  --     Unknown data action
  ErrUnknSymbol, --     Illegal memory symbol
  ErrLdIlgArg,
  ErrLdAddrRng,
  ErrLdMemInfo,
  ErrLdFileEmpty,
  ErrPrintString
);

TYPE VitalMemoryErrorSeverityType IS
ARRAY (VitalMemoryErrorType) OF SEVERITY_LEVEL;
CONSTANT VitalMemoryErrorSeverity :
  VitalMemoryErrorSeverityType := (
    ErrGoodAddr     => NOTE,
    ErrUnknAddr     => WARNING,
    ErrInvaAddr     => WARNING,
    ErrGoodTrAddr   => NOTE,
    ErrUnknTrAddr   => WARNING,
    ErrInvaTrAddr   => WARNING,
    ErrWrDatMem     => NOTE,
    ErrNoChgMem     => NOTE,
    ErrCrAllMem     => WARNING,
    ErrCrWrdMem     => WARNING,
    ErrCrBitMem     => WARNING,
    ErrCrDatMem     => WARNING,
    ErrCrAllSubMem  => WARNING,
    ErrCrWrdSubMem  => WARNING,
    ErrCrBitSubMem  => WARNING,
    ErrCrDatSubMem  => WARNING,
    ErrCrWrdOut     => WARNING,
    ErrCrBitOut     => WARNING,
    ErrCrDatOut     => WARNING,
    ErrCrWrdSubOut  => WARNING,
    ErrCrBitSubOut  => WARNING,
    ErrCrDatSubOut  => WARNING,
    ErrImplOut      => NOTE,
    ErrReadOut      => NOTE,
    ErrAssgOut      => NOTE,
    ErrAsgXOut      => NOTE,
    ErrAsg0Out      => NOTE,
    ErrAsg1Out      => NOTE,
    ErrAsgZOut      => NOTE,
    ErrAsgSOut      => NOTE,
    ErrAsgXMem      => NOTE,
    ErrAsg0Mem      => NOTE,
    ErrAsg1Mem      => NOTE,
    ErrAsgZMem      => NOTE,
    ErrDefMemAct    => NOTE,
    ErrInitMem      => NOTE,
    ErrMcpWrCont    => WARNING,
    ErrMcpCpCont    => WARNING,
    ErrMcpCpRead    => WARNING,
    ErrMcpRdWrCo    => WARNING,
    ErrMcpCpWrCont  => WARNING,
    ErrUnknMemDo    => ERROR,
    ErrUnknDatDo    => ERROR,
    ErrUnknSymbol   => ERROR,
    ErrLdIlgArg     => ERROR,
    ErrLdAddrRng    => WARNING,
    ErrLdMemInfo    => NOTE,
    ErrLdFileEmpty  => ERROR,
    ErrPrintString  => WARNING
  );

-- ----------------------------------------------------------------------------
CONSTANT MsgGoodAddr    : STRING
          := "Good address (no transition)";
CONSTANT MsgUnknAddr    : STRING
          := "Unknown address (no transition)";
CONSTANT MsgInvaAddr    : STRING
          := "Invalid address (no transition)";
CONSTANT MsgGoodTrAddr  : STRING
          := "Good address (with transition)";
CONSTANT MsgUnknTrAddr  : STRING
          := "Unknown address (with transition)";
CONSTANT MsgInvaTrAddr  : STRING
          := "Invalid address (with transition)";
CONSTANT MsgNoChgMem    : STRING
          := "Retaining previous memory contents";
CONSTANT MsgWrDatMem    : STRING
```

 Copyright © 2001 IEEE. All rights reserved.

```
            := “Writing data to memory”;
    CONSTANT MsgCrAllMem    : STRING
            := “Corrupting entire memory with ‘X’ ”;
    CONSTANT MsgCrWrdMem    : STRING
            := “Corrupting a word in memory with ‘X’ ”;
    CONSTANT MsgCrBitMem    : STRING
            := “Corrupting a single bit in memory with ‘X’ ”;
    CONSTANT MsgCrDatMem    : STRING
            := “Corrupting a word with ‘X’ based on data in”;
    CONSTANT MsgCrAllSubMem : STRING
            := “Corrupting a sub-word entire memory with ‘X’ ”;
    CONSTANT MsgCrWrdSubMem : STRING
            := “Corrupting a sub-word in memory with ‘X’ ”;
    CONSTANT MsgCrBitSubMem : STRING
            := “Corrupting a single bit of a sub-word with ‘X’ ”;
    CONSTANT MsgCrDatSubMem : STRING
            := “Corrupting a sub-word with ‘X’ based on data in”;
    CONSTANT MsgCrWrdOut    : STRING
            := “Corrupting data out with ‘X’ ”;
    CONSTANT MsgCrBitOut    : STRING
            := “Corrupting a single bit of data out with ‘X’ ”;
    CONSTANT MsgCrDatOut    : STRING
            := “Corrupting data out with ‘X’ based on data in”;
    CONSTANT MsgCrWrdSubOut : STRING
            := “Corrupting data out sub-word with ‘X’ ”;
    CONSTANT MsgCrBitSubOut : STRING
            := “Corrupting a single bit of data out sub-word with ‘X’ ”;
    CONSTANT MsgCrDatSubOut : STRING
            := “Corrupting data out sub-word with ‘X’ based on data in”;
    CONSTANT MsgImplOut     : STRING
            := “Implicit read from memory to data out”;
    CONSTANT MsgReadOut     : STRING
            := “Reading data from memory to data out”;
    CONSTANT MsgAssgOut     : STRING
            := “Transfering from data in to data out”;
    CONSTANT MsgAsgXOut     : STRING
            := “Assigning unknown level to data out”;
    CONSTANT MsgAsg0Out     : STRING
            := “Assigning low level to data out”;
    CONSTANT MsgAsg1Out     : STRING
            := “Assigning high level to data out”;
    CONSTANT MsgAsgZOut     : STRING
            := “Assigning high impedance to data out”;
    CONSTANT MsgAsgSOut     : STRING
            := “Keeping data out at steady value”;
    CONSTANT MsgAsgXMem     : STRING
            := “Assigning unknown level to memory location”;
    CONSTANT MsgAsg0Mem     : STRING
            := “Assigning low level to memory location”;
    CONSTANT MsgAsg1Mem     : STRING
            := “Assigning high level to memory location”;
    CONSTANT MsgAsgZMem     : STRING
            := “Assigning high impedance to memory location”;
    CONSTANT MsgDefMemAct   : STRING
            := “No memory table match, using default action”;
    CONSTANT MsgInitMem     : STRING
            := “Initializing memory contents”;
    CONSTANT MsgMcpWrCont   : STRING
            := “Same port write contention”;
    CONSTANT MsgMcpCpCont   : STRING
            := “Cross port read/write data/memory contention”;
    CONSTANT MsgMcpCpRead   : STRING
            := “Cross port read to same port”;
    CONSTANT MsgMcpRdWrCo   : STRING
            := “Cross port read/write data only contention”;
    CONSTANT MsgMcpCpWrCont : STRING
            := “Cross port write contention”;
    CONSTANT MsgUnknMemDo   : STRING
            := “Unknown memory action”;
    CONSTANT MsgUnknDatDo   : STRING
            := “Unknown data action”;
    CONSTANT MsgUnknSymbol  : STRING
            := “Illegal memory symbol”;

    CONSTANT MsgLdIlgArg    : STRING
            := “Illegal bit arguments while loading memory.”;
    CONSTANT MsgLdMemInfo   : STRING
            := “Loading data from the file into memory.”;
    CONSTANT MsgLdAddrRng   : STRING
            := “Address out of range while loading memory.”;
    CONSTANT MsgLdFileEmpty : STRING
```

Copyright © 2001 IEEE. All rights reserved.

```
          := "Memory load file is empty.";
CONSTANT MsgPrintString : STRING
          := "";

CONSTANT MsgUnknown     : STRING
          := "Unknown error message.";

CONSTANT MsgVMT         : STRING
          := "VitalMemoryTable";
CONSTANT MsgVMV         : STRING
          := "VitalMemoryViolation";
CONSTANT MsgVDM         : STRING
          := "VitalDeclareMemory";
CONSTANT MsgVMCP        : STRING
          := "VitalMemoryCrossPorts";

-- --------------------------------------------------------------------------------
-- LOCAL Utilities
-- --------------------------------------------------------------------------------

-- --------------------------------------------------------------------------------
-- Procedure:   MemoryMessage
-- Parameters:  ErrorId     -- Input error code
-- Description: This function looks up the input error code and returns
--              the string value of the associated message.
-- --------------------------------------------------------------------------------

FUNCTION MemoryMessage (
  CONSTANT ErrorId : IN VitalMemoryErrorType
) RETURN STRING IS
BEGIN
  CASE ErrorId IS
    WHEN ErrGoodAddr      => RETURN MsgGoodAddr   ;
    WHEN ErrUnknAddr      => RETURN MsgUnknAddr   ;
    WHEN ErrInvaAddr      => RETURN MsgInvaAddr   ;
    WHEN ErrGoodTrAddr    => RETURN MsgGoodTrAddr ;
    WHEN ErrUnknTrAddr    => RETURN MsgUnknTrAddr ;
    WHEN ErrInvaTrAddr    => RETURN MsgInvaTrAddr ;
    WHEN ErrWrDatMem      => RETURN MsgWrDatMem   ;
    WHEN ErrNoChgMem      => RETURN MsgNoChgMem   ;
    WHEN ErrCrAllMem      => RETURN MsgCrAllMem   ;
    WHEN ErrCrWrdMem      => RETURN MsgCrWrdMem   ;
    WHEN ErrCrBitMem      => RETURN MsgCrBitMem   ;
    WHEN ErrCrDatMem      => RETURN MsgCrDatMem   ;
    WHEN ErrCrAllSubMem   => RETURN MsgCrAllSubMem;
    WHEN ErrCrWrdSubMem   => RETURN MsgCrWrdSubMem;
    WHEN ErrCrBitSubMem   => RETURN MsgCrBitSubMem;
    WHEN ErrCrDatSubMem   => RETURN MsgCrDatSubMem;
    WHEN ErrCrWrdOut      => RETURN MsgCrWrdOut   ;
    WHEN ErrCrBitOut      => RETURN MsgCrBitOut   ;
    WHEN ErrCrDatOut      => RETURN MsgCrDatOut   ;
    WHEN ErrCrWrdSubOut   => RETURN MsgCrWrdSubOut;
    WHEN ErrCrBitSubOut   => RETURN MsgCrBitSubOut;
    WHEN ErrCrDatSubOut   => RETURN MsgCrDatSubOut;
    WHEN ErrImplOut       => RETURN MsgImplOut    ;
    WHEN ErrReadOut       => RETURN MsgReadOut    ;
    WHEN ErrAssgOut       => RETURN MsgAssgOut    ;
    WHEN ErrAsgXOut       => RETURN MsgAsgXOut    ;
    WHEN ErrAsg0Out       => RETURN MsgAsg0Out    ;
    WHEN ErrAsg1Out       => RETURN MsgAsg1Out    ;
    WHEN ErrAsgZOut       => RETURN MsgAsgZOut    ;
    WHEN ErrAsgSOut       => RETURN MsgAsgSOut    ;
    WHEN ErrAsgXMem       => RETURN MsgAsgXMem    ;
    WHEN ErrAsg0Mem       => RETURN MsgAsg0Mem    ;
    WHEN ErrAsg1Mem       => RETURN MsgAsg1Mem    ;
    WHEN ErrAsgZMem       => RETURN MsgAsgZMem    ;
    WHEN ErrDefMemAct     => RETURN MsgDefMemAct  ;
    WHEN ErrInitMem       => RETURN MsgInitMem    ;
    WHEN ErrMcpWrCont     => RETURN MsgMcpWrCont  ;
    WHEN ErrMcpCpCont     => RETURN MsgMcpCpCont  ;
    WHEN ErrMcpCpRead     => RETURN MsgMcpCpRead  ;
    WHEN ErrMcpRdWrCo     => RETURN MsgMcpRdWrCo  ;
    WHEN ErrMcpCpWrCont   => RETURN MsgMcpCpWrCont;
    WHEN ErrUnknMemDo     => RETURN MsgUnknMemDo  ;
    WHEN ErrUnknDatDo     => RETURN MsgUnknDatDo  ;
    WHEN ErrUnknSymbol    => RETURN MsgUnknSymbol ;
    WHEN ErrLdIlgArg      => RETURN MsgLdIlgArg   ;
    WHEN ErrLdAddrRng     => RETURN MsgLdAddrRng  ;
    WHEN ErrLdMemInfo     => RETURN MsgLdMemInfo  ;
    WHEN ErrLdFileEmpty   => RETURN MsgLdFileEmpty;
    WHEN ErrPrintString   => RETURN MsgPrintString;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    WHEN OTHERS           => RETURN MsgUnknown    ;
  END CASE;
END;

-- ------------------------------------------------------------------------------
-- Procedure:   PrintMemoryMessage
-- Parameters:  Routine     -- String identifying the calling routine
--                  ErrorId     -- Input error code for message lookup
--                  Info        -- Output string or character
--                  InfoStr     -- Additional output string
--                  Info1       -- Additional output integer
--                  Info2       -- Additional output integer
--                  Info3       -- Additional output integer
-- Description: This procedure prints out a memory status message
--              given the input error id and other status information.
-- ------------------------------------------------------------------------------
PROCEDURE PrintMemoryMessage (
  CONSTANT Routine : IN STRING;
  CONSTANT ErrorId : IN VitalMemoryErrorType
) IS
BEGIN
  ASSERT FALSE
    REPORT Routine & ": " & MemoryMessage(ErrorId)
    SEVERITY VitalMemoryErrorSeverity(ErrorId);
END;

-- ------------------------------------------------------------------------------
PROCEDURE PrintMemoryMessage (
  CONSTANT Routine : IN STRING;
  CONSTANT ErrorId : IN VitalMemoryErrorType;
  CONSTANT Info    : IN STRING
) IS
BEGIN
  ASSERT FALSE
    REPORT Routine & ": " & MemoryMessage(ErrorId) & " " & Info
    SEVERITY VitalMemoryErrorSeverity(ErrorId);
END;

-- ------------------------------------------------------------------------------
PROCEDURE PrintMemoryMessage (
  CONSTANT Routine : IN STRING;
  CONSTANT ErrorId : IN VitalMemoryErrorType;
  CONSTANT Info1   : IN STRING;
  CONSTANT Info2   : IN STRING
) IS
BEGIN
  ASSERT FALSE
    REPORT Routine & ": " & MemoryMessage(ErrorId) & " " & Info1 & " " & Info2
    SEVERITY VitalMemoryErrorSeverity(ErrorId);
END;

-- ------------------------------------------------------------------------------
PROCEDURE PrintMemoryMessage (
  CONSTANT Routine : IN STRING;
  CONSTANT ErrorId : IN VitalMemoryErrorType;
  CONSTANT Info    : IN CHARACTER
) IS
BEGIN
  ASSERT FALSE
    REPORT Routine & ": " & MemoryMessage(ErrorId) & " " & Info
    SEVERITY VitalMemoryErrorSeverity(ErrorId);
END;

-- ------------------------------------------------------------------------------
PROCEDURE PrintMemoryMessage (
  CONSTANT Routine    : IN STRING;
  CONSTANT ErrorId    : IN VitalMemoryErrorType;
  CONSTANT InfoStr    : IN STRING;
  CONSTANT Info1      : IN NATURAL
) IS
  VARIABLE TmpStr         : STRING ( 1 TO 256 ) ;
  VARIABLE TmpInt         : INTEGER := 1;
BEGIN
  IntToStr(Info1,TmpStr,TmpInt);
  ASSERT FALSE
    REPORT Routine & ": " & MemoryMessage(ErrorId) & " " & InfoStr & " " & TmpStr
    SEVERITY VitalMemoryErrorSeverity(ErrorId);
END;

-- ------------------------------------------------------------------------------
PROCEDURE PrintMemoryMessage (
```

Copyright © 2001 IEEE. All rights reserved.

```
    CONSTANT Routine    : IN STRING;
    CONSTANT ErrorId    : IN VitalMemoryErrorType;
    CONSTANT InfoStr    : IN STRING;
    CONSTANT Info1      : IN NATURAL;
    CONSTANT Info2      : IN NATURAL
  ) IS
    VARIABLE TmpStr         : STRING ( 1 TO 256 ) ;
    VARIABLE TmpInt         : INTEGER := 1;
  BEGIN
    IntToStr(Info1,TmpStr,TmpInt);
    IntToStr(Info2,TmpStr,TmpInt);
    ASSERT FALSE
      REPORT Routine & “: “ & MemoryMessage(ErrorId) & “ “ & InfoStr & “ “ & TmpStr
      SEVERITY VitalMemoryErrorSeverity(ErrorId);
  END;

  -- ------------------------------------------------------------------------------
  PROCEDURE PrintMemoryMessage (
    CONSTANT Routine    : IN STRING;
    CONSTANT ErrorId    : IN VitalMemoryErrorType;
    CONSTANT InfoStr    : IN STRING;
    CONSTANT Info1      : IN NATURAL;
    CONSTANT Info2      : IN NATURAL;
    CONSTANT Info3      : IN NATURAL
  ) IS
    VARIABLE TmpStr         : STRING ( 1 TO 256 ) ;
    VARIABLE TmpInt         : INTEGER := 1;
  BEGIN
    IntToStr(Info1,TmpStr,TmpInt);
    IntToStr(Info2,TmpStr,TmpInt);
    IntToStr(Info3,TmpStr,TmpInt);
    ASSERT FALSE
      REPORT Routine & “: “ & MemoryMessage(ErrorId) & “ “ & InfoStr & “ “ & TmpStr
      SEVERITY VitalMemoryErrorSeverity(ErrorId);
  END;

  -- ------------------------------------------------------------------------------
  PROCEDURE PrintMemoryMessage (
    CONSTANT Routine    : IN STRING;
    CONSTANT Table      : IN VitalMemoryTableType;
    CONSTANT Index      : IN INTEGER;
    CONSTANT InfoStr    : IN STRING
  ) IS
    CONSTANT TableEntries   : INTEGER := Table’LENGTH(1);
    CONSTANT TableWidth     : INTEGER := Table’LENGTH(2);
    VARIABLE TmpStr         : STRING ( 1 TO 256 ) ;
    VARIABLE TmpInt         : INTEGER := 1;
  BEGIN
    IF (Index < 0 AND Index > TableEntries-1) THEN
      ASSERT FALSE
        REPORT Routine & “: Memory table search failure”
        SEVERITY ERROR;
    END IF;
    ColLoop:
    FOR i IN 0 TO TableWidth-1 LOOP
      IF (i >= 64) THEN
        TmpStr(TmpInt) := ‘.’;
        TmpInt := TmpInt + 1;
        TmpStr(TmpInt) := ‘.’;
        TmpInt := TmpInt + 1;
        TmpStr(TmpInt) := ‘.’;
        TmpInt := TmpInt + 1;
        EXIT ColLoop;
      END IF;
      TmpStr(TmpInt) := ‘’’;
      TmpInt := TmpInt + 1;
      TmpStr(TmpInt) := To_MemoryChar(Table(Index,i));
      TmpInt := TmpInt + 1;
      TmpStr(TmpInt) := ‘’’;
      TmpInt := TmpInt + 1;
      IF (i < TableWidth-1) THEN
        TmpStr(TmpInt) := ‘,’;
        TmpInt := TmpInt + 1;
      END IF;
    END LOOP;
    ASSERT FALSE
      REPORT Routine & “: Port=” & InfoStr & “ TableRow=” & TmpStr
      SEVERITY NOTE;
  END;

  -- ------------------------------------------------------------------------------
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- Procedure:   DecodeAddress
-- Parameters:  Address        - Converted address.
--              AddrFlag       - Flag to indicte address match
--              MemoryData     - Information about memory characteristics
--              PrevAddressBus - Previous input address value
--              AddressBus     - Input address value.
-- Description: This procedure is used for transforming a valid
--              address value to an integer in order to access memory.
--              It performs address bound checking as well.
--              Sets Address to -1 for unknowns
--              Sets Address to -2 for out of range
-- --------------------------------------------------------------------------

PROCEDURE DecodeAddress (
  VARIABLE Address        : INOUT INTEGER;
  VARIABLE AddrFlag       : INOUT VitalMemorySymbolType;
  VARIABLE MemoryData     : IN VitalMemoryDataType;
  CONSTANT PrevAddressBus : IN std_logic_vector;
  CONSTANT AddressBus     : IN std_logic_vector
) IS
  VARIABLE Power      : NATURAL;
  VARIABLE AddrUnkn   : BOOLEAN;
BEGIN
  Power := 0;
  AddrUnkn := FALSE;
  -- It is assumed that always Address'LEFT represents the Most significant bit.
  FOR i IN AddressBus'RANGE LOOP
    Power := Power * 2;
    IF (AddressBus(i) /= '1' AND AddressBus(i) /= '0') THEN
      AddrUnkn := TRUE;
      Power := 0;
      EXIT;
    ELSIF (AddressBus(i) = '1') THEN
      Power := Power + 1;
    END IF;
  END LOOP;
  Address := Power;
  AddrFlag := 'g';
  IF (AddrUnkn) THEN
    AddrFlag := 'u';    -- unknown addr
    Address := -1;
  END IF;
  IF ( Power > (MemoryData.NoOfWords - 1)) THEN
    AddrFlag := 'i';    -- invalid addr
    Address := -2;
  END IF;
  IF (PrevAddressBus /= AddressBus) THEN
    CASE AddrFlag IS
      WHEN 'g'     => AddrFlag := 'G';
      WHEN 'u'     => AddrFlag := 'U';
      WHEN 'i'     => AddrFlag := 'I';
      WHEN OTHERS =>
        ASSERT FALSE REPORT
          "DecodeAddress: Internal error. [AddrFlag]="
          & To_MemoryChar(AddrFlag)
          SEVERITY ERROR;
    END CASE;
  END IF;
END DecodeAddress;

-- --------------------------------------------------------------------------
-- Procedure:   DecodeData
-- Parameters:  DataFlag      - Flag to indicte data match
--              PrevDataInBus - Previous input data value
--              DataInBus     - Input data value.
--              HighBit       - High bit offset value.
--              LowBit        - Low  bit offset value.
-- Description: This procedure is used for interpreting the input data
--              as a data flag for subsequent table matching.
-- --------------------------------------------------------------------------
PROCEDURE DecodeData (
  VARIABLE DataFlag       : INOUT VitalMemorySymbolType;
  CONSTANT PrevDataInBus  : IN std_logic_vector;
  CONSTANT DataInBus      : IN std_logic_vector;
  CONSTANT HighBit        : IN    NATURAL;
  CONSTANT LowBit         : IN    NATURAL
) IS
  VARIABLE DataUnkn   : BOOLEAN := FALSE;
BEGIN
  FOR i IN LowBit TO HighBit LOOP
    IF DataInBus(i) /= '1' AND DataInBus(i) /= '0' THEN
```

Copyright © 2001 IEEE. All rights reserved.

```
      DataUnkn := TRUE;
      EXIT;
    END IF;
  END LOOP;
  DataFlag := 'g';
  IF (DataUnkn) THEN
    DataFlag := 'u';    -- unknown addr
  END IF;
  IF (PrevDataInBus(HighBit DOWNTO LowBit) /=
      DataInBus(HighBit DOWNTO LowBit)) THEN
    CASE DataFlag IS
      WHEN 'g'    => DataFlag := 'G';
      WHEN 'u'    => DataFlag := 'U';
      WHEN OTHERS =>
        ASSERT FALSE REPORT
          "DecodeData: Internal error. [DataFlag]="
          & To_MemoryChar(DataFlag)
          SEVERITY ERROR;
    END CASE;
  END IF;
END DecodeData;

-- ----------------------------------------------------------------------------
-- Procedure:   WriteMemory
-- Parameters:  MemoryPtr   - Pointer to the memory array.
--              DataInBus   - Input Data to be written.
--              Address     - Address of the memory location.
--              BitPosition - Position of bit in memory location.
--              HighBit     - High bit offset value.
--              LowBit      - Low  bit offset value.
-- Description: This procedure is used to write to a memory location
--              on a bit/byte/word basis.
--              The high bit and low bit offset are used for byte write
--              operations.These parameters specify the data byte for write.
--              In the case of word write the complete memory word is used.
--              This procedure is overloaded for bit,byte and word write
--              memory operations.The number of parameters may vary.
-- ----------------------------------------------------------------------------
PROCEDURE WriteMemory (
  VARIABLE MemoryPtr   : INOUT VitalMemoryDataType;
  CONSTANT DataInBus   : IN    std_logic_vector;
  CONSTANT Address     : IN    INTEGER;
  CONSTANT HighBit     : IN    NATURAL;
  CONSTANT LowBit      : IN    NATURAL
) IS
  VARIABLE TmpData : std_logic_vector(DataInBus'LENGTH - 1 DOWNTO 0);
BEGIN
  -- Address bound checking.
  IF ( Address < 0 OR Address > (MemoryPtr.NoOfWords - 1)) THEN
    PrintMemoryMessage ( "WriteMemory", ErrPrintString,
      "Aborting write operation as address is out of range.") ;
    RETURN;
  END IF;
  TmpData  := To_UX01(DataInBus);
  FOR i in LowBit to HighBit LOOP
    MemoryPtr.MemoryArrayPtr(Address).all(i) := TmpData(i);
  END LOOP;
END WriteMemory;

-- ----------------------------------------------------------------------------
PROCEDURE WriteMemory (
  VARIABLE MemoryPtr      : INOUT VitalMemoryDataType;
  CONSTANT DataInBus      : IN    std_logic_vector;
  CONSTANT Address        : IN    INTEGER;
  CONSTANT BitPosition    : IN    NATURAL
) IS
  VARIABLE HighBit      : NATURAL;
  VARIABLE LowBit       : NATURAL;
BEGIN
  HighBit      := BitPosition;
  LowBit       := BitPosition;
  WriteMemory (MemoryPtr, DataInBus, Address, HighBit, LowBit);
END WriteMemory;

-- ----------------------------------------------------------------------------
PROCEDURE WriteMemory   (
  VARIABLE MemoryPtr   : INOUT VitalMemoryDataType;
  CONSTANT DataInBus   : IN    std_logic_vector;
  CONSTANT Address     : IN    INTEGER
) IS
  VARIABLE HighBit      : NATURAL;
```

 Copyright © 2001 IEEE. All rights reserved.

```
  VARIABLE LowBit       : NATURAL;
BEGIN
  HighBit      := MemoryPtr.NoOfBitsPerWord - 1;
  LowBit       := 0;
  WriteMemory (MemoryPtr, DataInBus, Address, HighBit, LowBit);
END WriteMemory;

-- ----------------------------------------------------------------------------
-- Procedure:   ReadMemory
-- Parameters:  MemoryPtr    - Pointer to the memory array.
--              DataOut      - Output Data to be read in this.
--              Address      - Address of the memory location.
--              BitPosition  - Position of bit in memory location.
--              HighBit      - High bit offset value.
--              LowBit       - Low  bit offset value.
-- Description: This procedure is used to read from a memory location
--              on a bit/byte/word basis.
--              The high bit and low bit offset are used for byte write
--              operations.These parameters specify the data byte for
--              read.In the case of word write the complete memory word
--              is used.This procedure is overloaded for bit,byte and
--              word write memory operations.The number of parameters
--              may vary.
-- ----------------------------------------------------------------------------
PROCEDURE ReadMemory (
  VARIABLE MemoryPtr  : INOUT VitalMemoryDataType;
  VARIABLE DataOut    : OUT   std_logic_vector;
  CONSTANT Address    : IN    INTEGER;
  CONSTANT HighBit    : IN    NATURAL;
  CONSTANT LowBit     : IN    NATURAL
) IS
  VARIABLE DataOutTmp : std_logic_vector(MemoryPtr.NoOfBitsPerWord-1 DOWNTO 0);
  VARIABLE length     : NATURAL := (HighBit - LowBit + 1);
BEGIN
  -- Address bound checking.
  IF ( Address > (MemoryPtr.NoOfWords - 1)) THEN
    PrintMemoryMessage (
      "ReadMemory",ErrInvaAddr,
      "[ Address,NoOfWords] =",Address,MemoryPtr.NoOfWords
    );
    FOR i in LowBit to HighBit LOOP
      DataOutTmp(i) := 'X';
    END LOOP;
  ELSE
    FOR i in LowBit to HighBit LOOP
      DataOutTmp(i) := MemoryPtr.MemoryArrayPtr (Address).all(i);
    END LOOP;
  END IF;
  DataOut :=  DataOutTmp;
END ReadMemory;

-- ----------------------------------------------------------------------------
PROCEDURE ReadMemory (
  VARIABLE MemoryPtr   : INOUT VitalMemoryDataType;
  VARIABLE DataOut     : OUT   std_logic_vector;
  CONSTANT Address     : IN    INTEGER;
  CONSTANT BitPosition : IN    NATURAL
) IS
  VARIABLE HighBit    : NATURAL;
  VARIABLE LowBit     : NATURAL;
BEGIN
  HighBit      := BitPosition;
  LowBit       := BitPosition;
  ReadMemory (MemoryPtr, DataOut, Address, HighBit, LowBit);
END ReadMemory;

-- ----------------------------------------------------------------------------
PROCEDURE ReadMemory (
  VARIABLE MemoryPtr  : INOUT VitalMemoryDataType;
  VARIABLE DataOut    : OUT   std_logic_vector;
  CONSTANT Address    : IN    INTEGER
) IS
  VARIABLE HighBit    : NATURAL;
  VARIABLE LowBit     : NATURAL;
BEGIN
  HighBit      := MemoryPtr.NoOfBitsPerWord - 1;
  LowBit       := 0;
  ReadMemory (MemoryPtr, DataOut, Address, HighBit, LowBit);
END ReadMemory;
```

Copyright © 2001 IEEE. All rights reserved.

```
-- ---------------------------------------------------------------------------
-- Procedure:   LoadMemory
-- Parameters:  MemoryPtr   - Pointer to the memory array.
--              FileName    - Name of the output file.
--              HighBit     - High bit offset value.
--              LowBit      - Low  bit offset value.
-- Description: This procedure is used to load the contents of the memory
--              from a specified input file.
--              The high bit and low bit offset are used so that same task
--              can be used for all bit/byte/word write operations.
--              In the case of a bit write RAM the HighBit and LowBit have
--              the same value.
--              This procedure is overloaded for word write operations.
-- ---------------------------------------------------------------------------
PROCEDURE LoadMemory (
  VARIABLE MemoryPtr  : INOUT VitalMemoryDataType;
  CONSTANT FileName   : IN    STRING;
  CONSTANT BinaryFile : IN BOOLEAN := FALSE
) IS
  FILE     Fptr      : TEXT OPEN read_mode IS FileName;
  VARIABLE OneLine   : LINE;
  VARIABLE Ignore    : CHARACTER;
  VARIABLE Index     : NATURAL  := 1;
  VARIABLE LineNo    : NATURAL  := 0;
  VARIABLE Address   : INTEGER  := 0;
  VARIABLE DataInBus : std_logic_vector(MemoryPtr.NoOfBitsPerWord-1  DOWNTO 0);
  VARIABLE AddrStr   : STRING(1 TO 80) ;
  VARIABLE DataInStr : STRING(1 TO 255) ;
BEGIN
  IF (ENDFILE(fptr)) THEN
    PrintMemoryMessage (MsgVDM, ErrLdFileEmpty,
      "[FileName]="&FileName);
    RETURN;
  END IF ;
  PrintMemoryMessage (
    MsgVDM,ErrLdMemInfo, "[FileName]="&FileName
  );
  WHILE (NOT ENDFILE(fptr)) LOOP
    ReadLine(Fptr, OneLine);
    LineNo := LineNo + 1 ;
    --  First ignoring leading spaces.
    WHILE (OneLine'LENGTH /= 0 and IsSpace(OneLine(1))) LOOP
      READ (OneLine, Ignore) ;     -- Ignoring the space character.
    END LOOP ;
    --  Note that, by now oneline has been "stripped" of its leading spaces.
    IF ( OneLine(1) = '@' ) THEN
      READ (OneLine, Ignore);  -- Ignore the '@' character and read the string.
      -- Now strip off spaces, if any, between '@' and Address string.
      WHILE (OneLine'LENGTH /= 0 and IsSpace(OneLine(1))) LOOP
        READ (OneLine, Ignore) ;     -- Ignoring the space character.
      END LOOP ;
      -- Now get the string which represents the address into string variable.
      Index := 1;
      WHILE (OneLine'LENGTH /= 0 AND (NOT(IsSpace(OneLine(1))))) LOOP
        READ(OneLine, AddrStr(Index));
        Index := Index + 1;
      END LOOP ;
      AddrStr(Index) := NUL;
      -- Now convert the hex string into a hex integer
      Address := HexToInt(AddrStr) ;
    ELSE
      IF ( LineNo /= 1 ) THEN
        Address := Address + 1;
      END IF;
    END IF ;
    IF ( Address > (MemoryPtr.NoOfWords - 1) ) THEN
      PrintMemoryMessage (MsgVDM, ErrLdAddrRng,
      "[Address,lineno]=", Address, LineNo) ;
      EXIT ;
    END IF;
    -- Now strip off spaces, between Address string and DataInBus string.
    WHILE (OneLine'LENGTH /= 0 AND IsSpace(OneLine(1))) LOOP
      READ (OneLine, Ignore) ;     -- Ignoring the space character.
    END LOOP ;
    Index := 1;
    WHILE (OneLine'LENGTH /= 0 AND (NOT(IsSpace(OneLine(1))))) LOOP
      READ(OneLine, DataInStr(Index));
      Index := Index + 1;
    END LOOP ;
    DataInStr(Index) := NUL;
    IF (BinaryFile) THEN
```

 Copyright © 2001 IEEE. All rights reserved.

```
      DataInBus := BinToBitv (DataInStr);
    ELSE
      DataInBus := HexToBitv (DataInStr);
    END IF ;
    WriteMemory (MemoryPtr, DataInBus, Address);
  END LOOP ;
END LoadMemory;

-- ----------------------------------------------------------------------------
-- Procedure:   MemoryMatch
-- Parameters:  Symbol       - Symbol from memory table
--              TestFlag     - Interpreted data or address symbol
--              In2          - input from VitalMemoryTable procedure
--                             to memory table
--              In2LastValue - Previous value of input
--              Err          - TRUE if symbol is not a valid input symbol
--              ReturnValue  - TRUE if match occurred
-- Description: This procedure sets ReturnValue to true if in2 matches
--              symbol (from the memory table).  If symbol is an edge
--              value edge is set to true and in2 and in2LastValue are
--              checked against symbol.  Err is set to true if symbol
--              is an invalid value for the input portion of the memory
--              table.
-- ----------------------------------------------------------------------------
PROCEDURE MemoryMatch (
  CONSTANT Symbol       : IN VitalMemorySymbolType;
  CONSTANT In2          : IN std_ulogic;
  CONSTANT In2LastValue : IN std_ulogic;
  VARIABLE Err          : OUT BOOLEAN;
  VARIABLE ReturnValue  : OUT BOOLEAN
) IS
BEGIN
  IF (NOT ValidMemoryTableInput(Symbol) ) THEN
    PrintMemoryMessage(MsgVMT,ErrUnknSymbol,To_MemoryChar(Symbol));
    Err := TRUE;
    ReturnValue := FALSE;
  ELSE
    ReturnValue := MemoryTableMatch(To_X01(In2LastValue), To_X01(In2), Symbol);
    Err := FALSE;
  END IF;
END;

-- ----------------------------------------------------------------------------
PROCEDURE MemoryMatch (
  CONSTANT Symbol       : IN VitalMemorySymbolType;
  CONSTANT TestFlag     : IN VitalMemorySymbolType;
  VARIABLE Err          : OUT BOOLEAN;
  VARIABLE ReturnValue  : OUT BOOLEAN
) IS
BEGIN
  Err := FALSE;
  ReturnValue := FALSE;
  CASE Symbol IS
    WHEN 'g' |' u' |' i' |' G' |' U' |' I' |' -' |' *' |' S'  =>
      IF (Symbol = TestFlag) THEN
        ReturnValue := TRUE;
      ELSE
        CASE Symbol IS
          WHEN '-'     =>
            ReturnValue := TRUE;
            Err := FALSE;
          WHEN '*'     =>
            IF (TestFlag = 'G' OR
                TestFlag = 'U' OR
                TestFlag = 'I' ) THEN
              ReturnValue := TRUE;
              Err := FALSE;
            END IF;
          WHEN 'S'     =>
            IF (TestFlag = 'g' OR
                TestFlag = 'u' OR
                TestFlag = 'i' ) THEN
                ReturnValue := TRUE;
              Err := FALSE;
            END IF;
          WHEN OTHERS =>
            ReturnValue := FALSE;
        END CASE;
      END IF;
    WHEN OTHERS =>
      Err := TRUE;
```

Copyright © 2001 IEEE. All rights reserved.

```
      RETURN;
    END CASE;
  END;

  -- ------------------------------------------------------------------------------
  -- Procedure:   MemoryTableCorruptMask
  -- Description: Compute memory and data corruption masks for memory table
  -- ------------------------------------------------------------------------------
  PROCEDURE MemoryTableCorruptMask (
    VARIABLE CorruptMask            : OUT std_logic_vector;
    CONSTANT Action                 : IN VitalMemorySymbolType;
    CONSTANT EnableIndex            : IN INTEGER;
    CONSTANT BitsPerWord            : IN INTEGER;
    CONSTANT BitsPerSubWord         : IN INTEGER;
    CONSTANT BitsPerEnable          : IN INTEGER
  ) IS
    VARIABLE CorruptMaskTmp : std_logic_vector (CorruptMask'RANGE)
                            := (OTHERS => '0');
    VARIABLE ViolFlAryPosn  : INTEGER;
    VARIABLE HighBit        : INTEGER;
    VARIABLE LowBit         : INTEGER;
  BEGIN
    CASE (Action) IS
      WHEN 'c'|'l'|'e'   =>
        -- Corrupt whole word
        CorruptMaskTmp := (OTHERS => 'X');
        CorruptMask := CorruptMaskTmp;
        RETURN;
      WHEN 'd'|'C'|'L'|'D'|'E'  =>
        -- Process corruption below
      WHEN OTHERS =>
        -- No data or memory corruption
        CorruptMaskTmp := (OTHERS => '0');
        CorruptMask := CorruptMaskTmp;
        RETURN;
    END CASE;
    IF (Action = 'd') THEN
      CorruptMaskTmp := (OTHERS => 'X');
      CorruptMask := CorruptMaskTmp;
      RETURN;
    END IF;
    -- Remaining are subword cases 'C', 'L', 'D', 'E'
    CorruptMaskTmp := (OTHERS => '0');
    LowBit := 0;
    HighBit := BitsPerSubWord-1;
    SubWordLoop:
    FOR i IN 0 TO BitsPerEnable-1 LOOP
      IF (i = EnableIndex) THEN
        FOR j IN HighBit TO LowBit LOOP
          CorruptMaskTmp(j) := 'X';
        END LOOP;
      END IF;
      -- Calculate HighBit and LowBit
      LowBit := LowBit + BitsPerSubWord;
      IF (LowBit > BitsPerWord) THEN
        LowBit := BitsPerWord;
      END IF;
      HighBit := LowBit + BitsPerSubWord;
      IF (HighBit > BitsPerWord) THEN
        HighBit := BitsPerWord;
      ELSE
        HighBit := HighBit - 1;
      END IF;
    END LOOP;
    CorruptMask := CorruptMaskTmp;
    RETURN;
  END;

  -- ------------------------------------------------------------------------------
  PROCEDURE MemoryTableCorruptMask (
    VARIABLE CorruptMask            : OUT std_logic_vector;
    CONSTANT Action                 : IN VitalMemorySymbolType
  ) IS
    VARIABLE CorruptMaskTmp : std_logic_vector (0 TO CorruptMask'LENGTH-1)
                            := (OTHERS => '0');
    VARIABLE ViolFlAryPosn  : INTEGER;
    VARIABLE HighBit        : INTEGER;
    VARIABLE LowBit         : INTEGER;
  BEGIN
    CASE (Action) IS
      WHEN 'c'|'l'|'d'|'e'|'C'|'L'|'D'|'E'  =>
```

 Copyright © 2001 IEEE. All rights reserved.

```
        -- Corrupt whole word
        CorruptMaskTmp := (OTHERS => 'X');
        CorruptMask := CorruptMaskTmp;
        RETURN;
      WHEN OTHERS =>
        -- No data or memory corruption
        CorruptMaskTmp := (OTHERS => '0');
        CorruptMask := CorruptMaskTmp;
        RETURN;
    END CASE;
    RETURN;
  END;

  -- ------------------------------------------------------------------------------
  -- Procedure:   MemoryTableCorruptMask
  -- Description: Compute memory and data corruption masks for violation table
  -- ------------------------------------------------------------------------------
  PROCEDURE ViolationTableCorruptMask (
    VARIABLE CorruptMask           : OUT std_logic_vector;
    CONSTANT Action                : IN VitalMemorySymbolType;
    CONSTANT ViolationFlags        : IN std_logic_vector;
    CONSTANT ViolationFlagsArray   : IN std_logic_vector;
    CONSTANT ViolationSizesArray   : IN VitalMemoryViolFlagSizeType;
    CONSTANT ViolationTable        : IN VitalMemoryTableType;
    CONSTANT TableIndex            : IN INTEGER;
    CONSTANT BitsPerWord           : IN INTEGER;
    CONSTANT BitsPerSubWord        : IN INTEGER;
    CONSTANT BitsPerEnable         : IN INTEGER
  ) IS
    VARIABLE CorruptMaskTmp : std_logic_vector (CorruptMask'RANGE)
                            := (OTHERS => '0');
    VARIABLE ViolMaskTmp    : std_logic_vector (CorruptMask'RANGE)
                            := (OTHERS => '0');
    VARIABLE ViolFlAryPosn  : INTEGER;
    VARIABLE HighBit        : INTEGER;
    VARIABLE LowBit         : INTEGER;
    CONSTANT ViolFlagsSize  : INTEGER := ViolationFlags'LENGTH;
    CONSTANT ViolFlArySize  : INTEGER := ViolationFlagsArray'LENGTH;
    CONSTANT TableEntries   : INTEGER := ViolationTable'LENGTH(1);
    CONSTANT TableWidth     : INTEGER := ViolationTable'LENGTH(2);
    CONSTANT DatActionNdx   : INTEGER := TableWidth - 1;
    CONSTANT MemActionNdx   : INTEGER := TableWidth - 2;
  BEGIN
    CASE (Action) IS
      WHEN 'c'|'l'|'e'   =>
        -- Corrupt whole word
        CorruptMaskTmp := (OTHERS => 'X');
        CorruptMask := CorruptMaskTmp;
        RETURN;
      WHEN 'd'|'C'|'L'|'D'|'E'  =>
        -- Process corruption below
      WHEN OTHERS =>
        -- No data or memory corruption
        CorruptMaskTmp := (OTHERS => '0');
        CorruptMask := CorruptMaskTmp;
        RETURN;
    END CASE;
    RowLoop: -- Check each element of the ViolationFlags
    FOR j IN 0 TO ViolFlagsSize LOOP
      IF (j = ViolFlagsSize) THEN
        ViolFlAryPosn := 0;
        RowLoop2: -- Check relevant elements of the ViolationFlagsArray
        FOR k IN 0 TO MemActionNdx - ViolFlagsSize - 1 LOOP
          IF (ViolationTable(TableIndex, k + ViolFlagsSize) = 'X') THEN
            MaskLoop: -- Set the 'X' bits in the violation mask
            FOR m IN 0 TO CorruptMask'LENGTH-1 LOOP
              IF (m <= ViolationSizesArray(k)-1) THEN
                ViolMaskTmp(m) := ViolMaskTmp(m) XOR
                ViolationFlagsArray(ViolFlAryPosn+m);
              ELSE
                EXIT MaskLoop;
              END IF;
            END LOOP;
          END IF;
          ViolFlAryPosn := ViolFlAryPosn + ViolationSizesArray(k);
        END LOOP;
      ELSE
        IF (ViolationTable(TableIndex, j) = 'X') THEN
          ViolMaskTmp(0) := ViolMaskTmp(0) XOR ViolationFlags(j);
        END IF;
      END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
      END LOOP;
      IF (Action = 'd') THEN
        CorruptMask := ViolMaskTmp;
        RETURN;
      END IF;
      -- Remaining are subword cases 'C', 'L', 'D', 'E'
      CorruptMaskTmp := (OTHERS => '0');
      LowBit := 0;
      HighBit := BitsPerSubWord-1;
      SubWordLoop:
      FOR i IN 0 TO BitsPerEnable-1 LOOP
        IF (ViolMaskTmp(i) = 'X') THEN
          FOR j IN HighBit TO LowBit LOOP
            CorruptMaskTmp(j) := 'X';
          END LOOP;
        END IF;
        -- Calculate HighBit and LowBit
        LowBit := LowBit + BitsPerSubWord;
        IF (LowBit > BitsPerWord) THEN
          LowBit := BitsPerWord;
        END IF;
        HighBit := LowBit + BitsPerSubWord;
        IF (HighBit > BitsPerWord) THEN
          HighBit := BitsPerWord;
        ELSE
          HighBit := HighBit - 1;
        END IF;
      END LOOP;
      CorruptMask := CorruptMaskTmp;
      RETURN;
    END;

    -- ------------------------------------------------------------------------------
    -- Procedure:   MemoryTableLookUp
    -- Parameters:  MemoryAction  - Output memory action to be performed
    --              DataAction    - Output data action to be performed
    --              PrevControls  - Previous data in for edge detection
    --              PrevEnableBus - Previous enables for edge detection
    --              Controls      - Agregate of scalar control lines
    --              EnableBus     - Concatenation of vector control lines
    --              EnableIndex   - Current slice of vector control lines
    --              AddrFlag      - Matching symbol from address decoding
    --              DataFlag      - Matching symbol from data decoding
    --              MemoryTable   - Input memory action table
    --              PortName      - Port name string for messages
    --              HeaderMsg     - Header string for messages
    --              MsgOn         - Control message output
    --
    -- Description: This function is used to find the output of the
    --              MemoryTable corresponding to a given set of inputs.
    --
    -- ------------------------------------------------------------------------------
    PROCEDURE MemoryTableLookUp (
      VARIABLE MemoryAction       : OUT VitalMemorySymbolType;
      VARIABLE DataAction         : OUT VitalMemorySymbolType;
      VARIABLE MemoryCorruptMask  : OUT std_logic_vector;
      VARIABLE DataCorruptMask    : OUT std_logic_vector;
      CONSTANT PrevControls       : IN std_logic_vector;
      CONSTANT Controls           : IN std_logic_vector;
      CONSTANT AddrFlag           : IN VitalMemorySymbolType;
      CONSTANT DataFlag           : IN VitalMemorySymbolType;
      CONSTANT MemoryTable        : IN VitalMemoryTableType;
      CONSTANT PortName           : IN STRING := "";
      CONSTANT HeaderMsg          : IN STRING := "";
      CONSTANT MsgOn              : IN BOOLEAN := TRUE
    ) IS
      CONSTANT ControlsSize   : INTEGER := Controls'LENGTH;
      CONSTANT TableEntries   : INTEGER := MemoryTable'LENGTH(1);
      CONSTANT TableWidth     : INTEGER := MemoryTable'LENGTH(2);
      CONSTANT DatActionNdx   : INTEGER := TableWidth - 1;
      CONSTANT MemActionNdx   : INTEGER := TableWidth - 2;
      CONSTANT DataInBusNdx   : INTEGER := TableWidth - 3;
      CONSTANT AddressBusNdx  : INTEGER := TableWidth - 4;
      VARIABLE AddrFlagTable  : VitalMemorySymbolType;
      VARIABLE Match  : BOOLEAN;
      VARIABLE Err    : BOOLEAN := FALSE;
      VARIABLE TableAlias     : VitalMemoryTableType(
                                  0 TO TableEntries - 1,
                                  0 TO TableWidth - 1)
                                  := MemoryTable;
    BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
   ColLoop: -- Compare each entry in the table
   FOR i IN TableAlias'RANGE(1) LOOP
     RowLoop: -- Check each element of the Controls
     FOR j IN 0 TO ControlsSize LOOP
       IF (j = ControlsSize) THEN
         -- a match occurred, now check AddrFlag, DataFlag
         MemoryMatch(TableAlias(i,AddressBusNdx),AddrFlag,Err,Match);
         IF (Match) THEN
           MemoryMatch(TableAlias(i,DataInBusNdx),DataFlag,Err,Match);
           IF (Match) THEN
             MemoryTableCorruptMask (
               CorruptMask     => MemoryCorruptMask    ,
               Action          => TableAlias(i, MemActionNdx)
             );
             MemoryTableCorruptMask (
               CorruptMask     => DataCorruptMask      ,
               Action          => TableAlias(i, DatActionNdx)
             );
             -- get the return memory and data actions
             MemoryAction := TableAlias(i, MemActionNdx);
             DataAction   := TableAlias(i, DatActionNdx);
             -- DEBUG: The lines below report table search
             IF (MsgOn) THEN
               PrintMemoryMessage(MsgVMT,TableAlias,i,PortName);
             END IF;
             -- DEBUG: The lines above report table search
             RETURN;
           END IF;
         END IF;
       ELSE
         -- Match memory table inputs
         MemoryMatch (   TableAlias(i,j),
         Controls(j), PrevControls(j),
         Err, Match);
       END IF;
       EXIT RowLoop WHEN NOT(Match);
       EXIT ColLoop WHEN Err;
     END LOOP RowLoop;
   END LOOP ColLoop;
   -- no match found, return default action
   MemoryAction := 's';   -- no change to memory
   DataAction   := 'S';   -- no change to dataout
   IF (MsgOn) THEN
   PrintMemoryMessage(MsgVMT,ErrDefMemAct,HeaderMsg,PortName);
   END IF;
   RETURN;
 END;

 -- -------------------------------------------------------------------------------
 PROCEDURE MemoryTableLookUp (
   VARIABLE MemoryAction       : OUT VitalMemorySymbolType;
   VARIABLE DataAction         : OUT VitalMemorySymbolType;
   VARIABLE MemoryCorruptMask  : OUT std_logic_vector;
   VARIABLE DataCorruptMask    : OUT std_logic_vector;
   CONSTANT PrevControls       : IN std_logic_vector;
   CONSTANT PrevEnableBus      : IN std_logic_vector;
   CONSTANT Controls           : IN std_logic_vector;
   CONSTANT EnableBus          : IN std_logic_vector;
   CONSTANT EnableIndex        : IN INTEGER;
   CONSTANT BitsPerWord        : IN INTEGER;
   CONSTANT BitsPerSubWord     : IN INTEGER;
   CONSTANT BitsPerEnable      : IN INTEGER;
   CONSTANT AddrFlag           : IN VitalMemorySymbolType;
   CONSTANT DataFlag           : IN VitalMemorySymbolType;
   CONSTANT MemoryTable        : IN VitalMemoryTableType;
   CONSTANT PortName           : IN STRING := "";
   CONSTANT HeaderMsg          : IN STRING := "";
   CONSTANT MsgOn              : IN BOOLEAN := TRUE
 ) IS
   CONSTANT ControlsSize   : INTEGER := Controls'LENGTH;
   CONSTANT TableEntries   : INTEGER := MemoryTable'LENGTH(1);
   CONSTANT TableWidth     : INTEGER := MemoryTable'LENGTH(2);
   CONSTANT DatActionNdx   : INTEGER := TableWidth - 1;
   CONSTANT MemActionNdx   : INTEGER := TableWidth - 2;
   CONSTANT DataInBusNdx   : INTEGER := TableWidth - 3;
   CONSTANT AddressBusNdx  : INTEGER := TableWidth - 4;
   VARIABLE AddrFlagTable  : VitalMemorySymbolType;
   VARIABLE Match  : BOOLEAN;
   VARIABLE Err    : BOOLEAN := FALSE;
   VARIABLE TableAlias     : VitalMemoryTableType(
                               0 TO TableEntries - 1,
```

Copyright © 2001 IEEE. All rights reserved.

```
                                0 TO TableWidth - 1)
                                  := MemoryTable;
    BEGIN
      ColLoop: -- Compare each entry in the table
      FOR i IN TableAlias'RANGE(1) LOOP
        RowLoop: -- Check each element of the Controls
        FOR j IN 0 TO ControlsSize LOOP
          IF (j = ControlsSize) THEN
            -- a match occurred, now check EnableBus, AddrFlag, DataFlag
            IF (EnableIndex >= 0) THEN
              RowLoop2: -- Check relevant elements of the EnableBus
              FOR k IN 0 TO AddressBusNdx - ControlsSize - 1 LOOP
                MemoryMatch (   TableAlias(i,k + ControlsSize),
                                EnableBus(k * BitsPerEnable + EnableIndex),
                                PrevEnableBus(k * BitsPerEnable + EnableIndex),
                                Err, Match);
                EXIT RowLoop2 WHEN NOT(Match);
              END LOOP;
            END IF;
            IF (Match) THEN
              MemoryMatch(TableAlias(i,AddressBusNdx),AddrFlag,Err,Match);
              IF (Match) THEN
                MemoryMatch(TableAlias(i,DataInBusNdx),DataFlag,Err,Match);
                IF (Match) THEN
                  MemoryTableCorruptMask (
                    CorruptMask     => MemoryCorruptMask    ,
                    Action          => TableAlias(i, MemActionNdx),
                    EnableIndex     => EnableIndex          ,
                    BitsPerWord     => BitsPerWord          ,
                    BitsPerSubWord  => BitsPerSubWord       ,
                    BitsPerEnable   => BitsPerEnable
                  );
                  MemoryTableCorruptMask (
                    CorruptMask     => DataCorruptMask      ,
                    Action          => TableAlias(i, DatActionNdx),
                    EnableIndex     => EnableIndex          ,
                    BitsPerWord     => BitsPerWord          ,
                    BitsPerSubWord  => BitsPerSubWord       ,
                    BitsPerEnable   => BitsPerEnable
                  );
                  -- get the return memory and data actions
                  MemoryAction := TableAlias(i, MemActionNdx);
                  DataAction   := TableAlias(i, DatActionNdx);
                  -- DEBUG: The lines below report table search
                  IF (MsgOn) THEN
                    PrintMemoryMessage(MsgVMT,TableAlias,i,PortName);
                  END IF;
                  -- DEBUG: The lines above report table search
                  RETURN;
                END IF;
              END IF;
            END IF;
          ELSE
            -- Match memory table inputs
            MemoryMatch (   TableAlias(i,j),
                            Controls(j), PrevControls(j),
                            Err, Match);
          END IF;
          EXIT RowLoop WHEN NOT(Match);
          EXIT ColLoop WHEN Err;
        END LOOP RowLoop;
      END LOOP ColLoop;
      -- no match found, return default action
      MemoryAction := 's';   -- no change to memory
      DataAction   := 'S';   -- no change to dataout
      IF (MsgOn) THEN
        PrintMemoryMessage(MsgVMT,ErrDefMemAct,HeaderMsg,PortName);
      END IF;
      RETURN;
    END;

    -- ------------------------------------------------------------------------------
    -- Procedure:   ViolationTableLookUp
    -- Parameters:  MemoryAction  - Output memory action to be performed
    --              DataAction    - Output data action to be performed
    --              TimingDataArray   - This is currently not used (comment out)
    --              ViolationArray    - Aggregation of violation variables
    --              ViolationTable    - Input memory violation table
    --              PortName      - Port name string for messages
    --              HeaderMsg     - Header string for messages
    --              MsgOn         - Control message output
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- Description: This function is used to find the output of the
--              ViolationTable corresponding to a given set of inputs.
-- -------------------------------------------------------------------------
PROCEDURE ViolationTableLookUp (
  VARIABLE MemoryAction           : OUT VitalMemorySymbolType;
  VARIABLE DataAction             : OUT VitalMemorySymbolType;
  VARIABLE MemoryCorruptMask      : OUT std_logic_vector;
  VARIABLE DataCorruptMask        : OUT std_logic_vector;
  CONSTANT ViolationFlags         : IN std_logic_vector;
  CONSTANT ViolationFlagsArray    : IN std_logic_vector;
  CONSTANT ViolationSizesArray    : IN VitalMemoryViolFlagSizeType;
  CONSTANT ViolationTable         : IN VitalMemoryTableType;
  CONSTANT BitsPerWord            : IN INTEGER;
  CONSTANT BitsPerSubWord         : IN INTEGER;
  CONSTANT BitsPerEnable          : IN INTEGER;
  CONSTANT PortName               : IN STRING := "";
  CONSTANT HeaderMsg              : IN STRING := "";
  CONSTANT MsgOn                  : IN BOOLEAN := TRUE
) IS
  CONSTANT ViolFlagsSize  : INTEGER := ViolationFlags'LENGTH;
  CONSTANT ViolFlArySize  : INTEGER := ViolationFlagsArray'LENGTH;
  VARIABLE ViolFlAryPosn  : INTEGER;
  VARIABLE ViolFlAryItem  : std_ulogic;
  CONSTANT ViolSzArySize  : INTEGER := ViolationSizesArray'LENGTH;
  CONSTANT TableEntries   : INTEGER := ViolationTable'LENGTH(1);
  CONSTANT TableWidth     : INTEGER := ViolationTable'LENGTH(2);
  CONSTANT DatActionNdx   : INTEGER := TableWidth - 1;
  CONSTANT MemActionNdx   : INTEGER := TableWidth - 2;
  VARIABLE HighBit        : NATURAL := 0;
  VARIABLE LowBit         : NATURAL := 0;
  VARIABLE Match  : BOOLEAN;
  VARIABLE Err    : BOOLEAN := FALSE;
  VARIABLE TableAlias     : VitalMemoryTableType(
                              0 TO TableEntries - 1,
                              0 TO TableWidth - 1)
                                := ViolationTable;
BEGIN
  ColLoop: -- Compare each entry in the table
  FOR i IN TableAlias'RANGE(1) LOOP
    RowLoop: -- Check each element of the ViolationFlags
    FOR j IN 0 TO ViolFlagsSize LOOP
      IF (j = ViolFlagsSize) THEN
        ViolFlAryPosn := 0;
        RowLoop2: -- Check relevant elements of the ViolationFlagsArray
        FOR k IN 0 TO MemActionNdx - ViolFlagsSize - 1 LOOP
          ViolFlAryItem := '0';
          SubwordLoop: -- Check for 'X' in ViolationFlagsArray chunk
          FOR s IN ViolFlAryPosn TO ViolFlAryPosn+ViolationSizesArray(k)-1 LOOP
            IF (ViolationFlagsArray(s) = 'X') THEN
              ViolFlAryItem := 'X';
              EXIT SubwordLoop;
            END IF;
          END LOOP;
          MemoryMatch (   TableAlias(i,k + ViolFlagsSize),
          ViolFlAryItem,ViolFlAryItem,
          Err, Match);
          ViolFlAryPosn := ViolFlAryPosn + ViolationSizesArray(k);
          EXIT RowLoop2 WHEN NOT(Match);
        END LOOP;
        IF (Match) THEN
          -- Compute memory and data corruption masks
          ViolationTableCorruptMask(
            CorruptMask            => MemoryCorruptMask    ,
            Action                 => TableAlias(i, MemActionNdx),
            ViolationFlags         => ViolationFlags       ,
            ViolationFlagsArray    => ViolationFlagsArray  ,
            ViolationSizesArray    => ViolationSizesArray  ,
            ViolationTable         => ViolationTable       ,
            TableIndex             => i                    ,
            BitsPerWord            => BitsPerWord          ,
            BitsPerSubWord         => BitsPerSubWord       ,
            BitsPerEnable          => BitsPerEnable
          );
          ViolationTableCorruptMask(
            CorruptMask            => DataCorruptMask      ,
            Action                 => TableAlias(i, DatActionNdx),
            ViolationFlags         => ViolationFlags       ,
            ViolationFlagsArray    => ViolationFlagsArray  ,
            ViolationSizesArray    => ViolationSizesArray  ,
            ViolationTable         => ViolationTable       ,
            TableIndex             => i                    ,
```

Copyright © 2001 IEEE. All rights reserved.

```
            BitsPerWord             => BitsPerWord          ,
            BitsPerSubWord          => BitsPerSubWord       ,
            BitsPerEnable           => BitsPerEnable
          );
          -- get the return memory and data actions
          MemoryAction := TableAlias(i, MemActionNdx);
          DataAction   := TableAlias(i, DatActionNdx);
          -- DEBUG: The lines below report table search
          IF (MsgOn) THEN
            PrintMemoryMessage(MsgVMV,TableAlias,i,PortName);
          END IF;
          -- DEBUG: The lines above report table search
          RETURN;
        END IF;
      ELSE
        -- Match violation table inputs
        Err := FALSE;
        Match := FALSE;
        IF (TableAlias(i,j) /= 'X' AND
            TableAlias(i,j) /= '0' AND
            TableAlias(i,j) /= '-') THEN
          Err := TRUE;
        ELSIF (TableAlias(i,j) = '-' OR
            (TableAlias(i,j) = 'X' AND ViolationFlags(j) = 'X') OR
            (TableAlias(i,j) = '0' AND ViolationFlags(j) = '0')) THEN
          Match := TRUE;
        END IF;
      END IF;
      EXIT RowLoop WHEN NOT(Match);
      EXIT ColLoop WHEN Err;
    END LOOP RowLoop;
  END LOOP ColLoop;
  -- no match found, return default action
  MemoryAction := 's';   -- no change to memory
  DataAction   := 'S';   -- no change to dataout
  IF (MsgOn) THEN
    PrintMemoryMessage(MsgVMV,ErrDefMemAct,HeaderMsg,PortName);
  END IF;
  RETURN;
END;

-- ------------------------------------------------------------------------------
-- Procedure:   HandleMemoryAction
-- Parameters:  MemoryData   - Pointer to memory data structure
--              PortFlag     - Indicates read/write mode of port
--              CorruptMask  - XOR'ed with DataInBus when corrupting
--              DataInBus    - Current data bus in
--              Address      - Current address integer
--              HighBit      - Current address high bit
--              LowBit       - Current address low bit
--              MemoryTable  - Input memory action table
--              MemoryAction - Memory action to be performed
--              PortName     - Port name string for messages
--              HeaderMsg    - Header string for messages
--              MsgOn        - Control message output
-- Description: This procedure performs the specified memory action on
--              the input memory data structure.
-- ------------------------------------------------------------------------------
PROCEDURE HandleMemoryAction (
  VARIABLE MemoryData     : INOUT VitalMemoryDataType;
  VARIABLE PortFlag       : INOUT VitalPortFlagType;
  CONSTANT CorruptMask    : IN std_logic_vector;
  CONSTANT DataInBus      : IN std_logic_vector;
  CONSTANT Address        : IN INTEGER;
  CONSTANT HighBit        : IN NATURAL;
  CONSTANT LowBit         : IN NATURAL;
  CONSTANT MemoryTable    : IN VitalMemoryTableType;
  CONSTANT MemoryAction   : IN VitalMemorySymbolType;
  CONSTANT CallerName     : IN STRING;
  CONSTANT PortName       : IN STRING := "";
  CONSTANT HeaderMsg      : IN STRING := "";
  CONSTANT MsgOn          : IN BOOLEAN := TRUE
) IS
  VARIABLE DataInTmp : std_logic_vector(DataInBus'RANGE)
                       := DataInBus;
  BEGIN

  -- Handle the memory action
  CASE MemoryAction IS

  WHEN 'w'     =>
```

 Copyright © 2001 IEEE. All rights reserved.

```
      -- Writing data to memory
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrWrDatMem,HeaderMsg,PortName);
      END IF;
      WriteMemory(MemoryData,DataInBus,Address,HighBit,LowBit);
      PortFlag.MemoryCurrent := WRITE;

    WHEN 's'     =>
      -- Retaining previous memory contents
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrNoChgMem,HeaderMsg,PortName);
      END IF;
      -- Set memory current to quiet state
      PortFlag.MemoryCurrent := READ;

    WHEN 'c'     =>
      -- Corrupting entire memory with 'X'
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrAllMem,HeaderMsg,PortName);
      END IF;
      DataInTmp := (OTHERS => 'X');
      -- No need to CorruptMask
      FOR i IN 0 TO MemoryData.NoOfWords-1 LOOP
        WriteMemory(MemoryData,DataInTmp,i);
      END LOOP;
      PortFlag.MemoryCurrent := CORRUPT;

    WHEN 'l'     =>
      -- Corrupting a word in memory with 'X'
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrWrdMem,HeaderMsg,PortName);
      END IF;
      DataInTmp := (OTHERS => 'X');
      -- No need to CorruptMask
      WriteMemory(MemoryData,DataInTmp,Address);
      PortFlag.MemoryCurrent := CORRUPT;

    WHEN 'd'     =>
      -- Corrupting a single bit in memory with 'X'
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrBitMem,HeaderMsg,PortName);
      END IF;
      ReadMemory(MemoryData,DataInTmp,Address);
      DataInTmp := DataInTmp XOR CorruptMask;
      WriteMemory(MemoryData,DataInTmp,Address,HighBit,LowBit);
      PortFlag.MemoryCurrent := CORRUPT;

    WHEN 'e'     =>
      -- Corrupting a word with 'X' based on data in
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrDatMem,HeaderMsg,PortName);
      END IF;
      ReadMemory(MemoryData,DataInTmp,Address);
      IF (DataInTmp /= DataInBus) THEN
        DataInTmp := (OTHERS => 'X');
        -- No need to CorruptMask
        WriteMemory(MemoryData,DataInTmp,Address);
      END IF;
      PortFlag.MemoryCurrent := CORRUPT;

    WHEN 'C'     =>
      -- Corrupting a sub-word entire memory with 'X'
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrAllSubMem,HeaderMsg,PortName);
      END IF;
      FOR i IN 0 TO MemoryData.NoOfWords-1 LOOP
        ReadMemory(MemoryData,DataInTmp,i);
        DataInTmp := DataInTmp XOR CorruptMask;
        WriteMemory(MemoryData,DataInTmp,i,HighBit,LowBit);
      END LOOP;
      PortFlag.MemoryCurrent := CORRUPT;

    WHEN 'L'     =>
      -- Corrupting a sub-word in memory with 'X'
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrWrdSubMem,HeaderMsg,PortName);
      END IF;
      ReadMemory(MemoryData,DataInTmp,Address);
      DataInTmp := DataInTmp XOR CorruptMask;
      WriteMemory(MemoryData,DataInTmp,Address,HighBit,LowBit);
      PortFlag.MemoryCurrent := CORRUPT;
```

Copyright © 2001 IEEE. All rights reserved.

```
    WHEN 'D'     =>
      -- Corrupting a single bit of a memory sub-word with 'X'
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrBitSubMem,HeaderMsg,PortName);
      END IF;
      ReadMemory(MemoryData,DataInTmp,Address);
      DataInTmp := DataInTmp XOR CorruptMask;
      WriteMemory(MemoryData,DataInTmp,Address,HighBit,LowBit);
      PortFlag.MemoryCurrent := CORRUPT;

    WHEN 'E'     =>
      -- Corrupting a sub-word with 'X' based on data in
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrCrDatSubMem,HeaderMsg,PortName);
      END IF;
      ReadMemory(MemoryData,DataInTmp,Address);
      IF (DataInBus(HighBit DOWNTO LowBit) /=
        DataInTmp(HighBit DOWNTO LowBit)) THEN
        DataInTmp(HighBit DOWNTO LowBit) := (OTHERS => 'X');
        WriteMemory(MemoryData,DataInTmp,Address,HighBit,LowBit);
      END IF;
      --PortFlag := WRITE;
      PortFlag.MemoryCurrent := CORRUPT;

    WHEN '0'     =>
      -- Assigning low level to memory location
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAsg0Mem,HeaderMsg,PortName);
      END IF;
      DataInTmp := (OTHERS => '0');
      WriteMemory(MemoryData,DataInTmp,Address, HighBit, LowBit);
      PortFlag.MemoryCurrent := WRITE;

    WHEN '1'     =>
      -- Assigning high level to memory location
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAsg1Mem,HeaderMsg,PortName);
      END IF;
      DataInTmp := (OTHERS => '1');
      WriteMemory(MemoryData,DataInTmp,Address, HighBit, LowBit);
      PortFlag.MemoryCurrent := WRITE;

    WHEN 'Z'     =>
      -- Assigning high impedence to memory location
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAsgZMem,HeaderMsg,PortName);
      END IF;
      DataInTmp := (OTHERS => 'Z');
      WriteMemory(MemoryData,DataInTmp,Address, HighBit, LowBit);
      PortFlag.MemoryCurrent := WRITE;

    WHEN OTHERS =>
      -- Unknown memory action
      PortFlag.MemoryCurrent := UNDEF;
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrUnknMemDo,HeaderMsg,PortName);
      END IF;

    END CASE;

    -- Note: HandleMemoryAction does not change the PortFlag.OutputDisable
  END;

  -- ----------------------------------------------------------------------------
  -- Procedure:   HandleDataAction
  -- Parameters:  DataOutBus  - Output result of the data action
  --              MemoryData  - Input pointer to memory data structure
  --              PortFlag    - Indicates read/write mode of port
  --              CorruptMask - XOR'ed with DataInBus when corrupting
  --              DataInBus   - Current data bus in
  --              Address     - Current address integer
  --              HighBit     - Current address high bit
  --              LowBit      - Current address low bit
  --              MemoryTable - Input memory action table
  --              DataAction  - Data action to be performed
  --              PortName    - Port name string for messages
  --              HeaderMsg   - Header string for messages
  --              MsgOn       - Control message output
  -- Description: This procedure performs the specified data action based
  --              on the input memory data structure.  Checks whether
```

 Copyright © 2001 IEEE. All rights reserved.

```
--              the previous state is HighZ. If yes then portFlag
--              should be NOCHANGE for VMPD to ignore IORetain
--              corruption. The idea is that the first Z should be
--              propagated but later ones should be ignored.
-- ----------------------------------------------------------------------------
PROCEDURE HandleDataAction (
  VARIABLE DataOutBus     : INOUT std_logic_vector;
  VARIABLE MemoryData     : INOUT VitalMemoryDataType;
  VARIABLE PortFlag       : INOUT VitalPortFlagType;
  CONSTANT CorruptMask    : IN std_logic_vector;
  CONSTANT DataInBus      : IN std_logic_vector;
  CONSTANT Address        : IN INTEGER;
  CONSTANT HighBit        : IN NATURAL;
  CONSTANT LowBit         : IN NATURAL;
  CONSTANT MemoryTable    : IN VitalMemoryTableType;
  CONSTANT DataAction     : IN VitalMemorySymbolType;
  CONSTANT CallerName     : IN STRING;
  CONSTANT PortName       : IN STRING := "";
  CONSTANT HeaderMsg      : IN STRING := "";
  CONSTANT MsgOn          : IN BOOLEAN := TRUE
) IS

  VARIABLE DataOutTmp : std_logic_vector(DataOutBus'RANGE)
                        := DataOutBus;

BEGIN

  -- Handle the data action
  CASE DataAction IS

  WHEN 'l'     =>
    -- Corrupting data out with 'X'
    IF (MsgOn) THEN
      PrintMemoryMessage(CallerName,ErrCrWrdOut,HeaderMsg,PortName);
    END IF;
    DataOutTmp := (OTHERS => 'X');
    -- No need to CorruptMask
    PortFlag.DataCurrent := CORRUPT;

  WHEN 'd'     =>
    -- Corrupting a single bit of data out with 'X'
    IF (MsgOn) THEN
      PrintMemoryMessage(CallerName,ErrCrBitOut,HeaderMsg,PortName);
    END IF;
    DataOutTmp(HighBit DOWNTO LowBit) :=
      DataOutTmp(HighBit DOWNTO LowBit) XOR
      CorruptMask(HighBit DOWNTO LowBit);
    PortFlag.DataCurrent := CORRUPT;

  WHEN 'e'     =>
    -- Corrupting data out with 'X' based on data in
    IF (MsgOn) THEN
      PrintMemoryMessage(CallerName,ErrCrDatOut,HeaderMsg,PortName);
    END IF;
    ReadMemory(MemoryData,DataOutTmp,Address);
    IF (DataOutTmp /= DataInBus) THEN
      DataOutTmp := (OTHERS => 'X');
      -- No need to CorruptMask
    END IF;
    PortFlag.DataCurrent := CORRUPT;

  WHEN 'L'     =>
    -- Corrupting data out sub-word with 'X'
    IF (MsgOn) THEN
      PrintMemoryMessage(CallerName,ErrCrWrdSubOut,HeaderMsg,PortName);
    END IF;
    ReadMemory(MemoryData,DataOutTmp,Address);
    DataOutTmp(HighBit DOWNTO LowBit) :=
      DataOutTmp(HighBit DOWNTO LowBit) XOR
      CorruptMask(HighBit DOWNTO LowBit);
    PortFlag.DataCurrent := CORRUPT;

  WHEN 'D'     =>
    -- Corrupting a single bit of data out sub-word with 'X'
    IF (MsgOn) THEN
      PrintMemoryMessage(CallerName,ErrCrBitSubOut,HeaderMsg,PortName);
    END IF;
    DataOutTmp(HighBit DOWNTO LowBit) :=
      DataOutTmp(HighBit DOWNTO LowBit) XOR
      CorruptMask(HighBit DOWNTO LowBit);
    PortFlag.DataCurrent := CORRUPT;
```

Copyright © 2001 IEEE. All rights reserved.

```
    WHEN 'E'     =>
      -- Corrupting data out sub-word with 'X' based on data in
      IF (MsgOn) THEN
      PrintMemoryMessage(CallerName,ErrCrDatSubOut,HeaderMsg,PortName);
      END IF;
      ReadMemory(MemoryData,DataOutTmp,Address);
      IF (DataInBus(HighBit DOWNTO LowBit) /=
          DataOutTmp(HighBit DOWNTO LowBit)) THEN
        DataOutTmp(HighBit DOWNTO LowBit) := (OTHERS => 'X');
        -- No need to CorruptMask
      END IF;
      PortFlag.DataCurrent := CORRUPT;

    WHEN 'M'     =>
      -- Implicit read from memory to data out
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrImplOut,HeaderMsg,PortName);
      END IF;
      PortFlag.DataCurrent := READ;

    WHEN 'm'     =>
      -- Reading data from memory to data out
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrReadOut,HeaderMsg,PortName);
      END IF;
      ReadMemory(MemoryData,DataOutTmp,Address);
      PortFlag.DataCurrent := READ;

    WHEN 't'     =>
      -- Transfering from data in to data out
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAssgOut,HeaderMsg,PortName);
      END IF;
      DataOutTmp := DataInBus;
      PortFlag.DataCurrent := READ;

    WHEN '0'     =>
      -- Assigning low level to data out
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAsg0Out,HeaderMsg,PortName);
      END IF;
      DataOutTmp := (OTHERS => '0');
      PortFlag.DataCurrent := READ;

    WHEN '1'     =>
      -- Assigning high level to data out
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAsg1Out,HeaderMsg,PortName);
      END IF;
      DataOutTmp := (OTHERS => '1');
      PortFlag.DataCurrent := READ;

    WHEN 'Z'     =>
      -- Assigning high impedence to data out
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAsgZOut,HeaderMsg,PortName);
      END IF;
      DataOutTmp := (OTHERS => 'Z');
      PortFlag.DataCurrent := HIGHZ;

    WHEN 'S'     =>
      -- Keeping data out at steady value
      PortFlag.OutputDisable := TRUE;
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrAsgSOut,HeaderMsg,PortName);
      END IF;

    WHEN OTHERS =>
      -- Unknown data action
      PortFlag.DataCurrent := UNDEF;
      IF (MsgOn) THEN
        PrintMemoryMessage(CallerName,ErrUnknDatDo,HeaderMsg,PortName);
      END IF;

    END CASE;

    DataOutBus(HighBit DOWNTO LowBit) := DataOutTmp(HighBit DOWNTO LowBit);

  END;
```


Copyright © 2001 IEEE. All rights reserved.

```
-- ------------------------------------------------------------------------
-- Memory Table Modeling Primitives
-- ------------------------------------------------------------------------

-- ------------------------------------------------------------------------
-- Procedure:   VitalDeclareMemory
-- Parameters:  NoOfWords          - Number of words in the memory
--              NoOfBitsPerWord    - Number of bits per word in memory
--              NoOfBitsPerSubWord - Number of bits per sub word
--              MemoryLoadFile     - Name of data file to load
-- Description: This function is intended to be used to initialize
--              memory data declarations, i.e. to be executed duing
--              simulation elaboration time.  Handles the allocation
--              and initialization of memory for the memory data.
--              Default NoOfBitsPerSubWord is NoOfBitsPerWord.
-- ------------------------------------------------------------------------
FUNCTION VitalDeclareMemory (
  CONSTANT NoOfWords          : IN POSITIVE;
  CONSTANT NoOfBitsPerWord    : IN POSITIVE;
  CONSTANT MemoryLoadFile     : IN string := "";
  CONSTANT BinaryLoadFile     : IN BOOLEAN := FALSE
) RETURN VitalMemoryDataType IS
  VARIABLE MemoryPtr      : VitalMemoryDataType;
BEGIN
  MemoryPtr := VitalDeclareMemory(
    NoOfWords           => NoOfWords,
    NoOfBitsPerWord     => NoOfBitsPerWord,
    NoOfBitsPerSubWord  => NoOfBitsPerWord,
    MemoryLoadFile      => MemoryLoadFile,
    BinaryLoadFile      => BinaryLoadFile
  );
  RETURN MemoryPtr;
END;

-- ------------------------------------------------------------------------
FUNCTION VitalDeclareMemory (
  CONSTANT NoOfWords          : IN POSITIVE;
  CONSTANT NoOfBitsPerWord    : IN POSITIVE;
  CONSTANT NoOfBitsPerSubWord : IN POSITIVE;
  CONSTANT MemoryLoadFile     : IN string := "";
  CONSTANT BinaryLoadFile     : IN BOOLEAN := FALSE
) RETURN VitalMemoryDataType IS
  VARIABLE MemoryPtr      : VitalMemoryDataType;
  VARIABLE BitsPerEnable  : NATURAL
                          := ((NoOfBitsPerWord-1)
                              /NoOfBitsPerSubWord)+1;
BEGIN
  PrintMemoryMessage(MsgVDM,ErrInitMem);
  MemoryPtr := new VitalMemoryArrayRecType '(
      NoOfWords           => NoOfWords,
      NoOfBitsPerWord     => NoOfBitsPerWord,
      NoOfBitsPerSubWord  => NoOfBitsPerSubWord,
      NoOfBitsPerEnable   => BitsPerEnable,
      MemoryArrayPtr      => NULL
    );
  MemoryPtr.MemoryArrayPtr
    := new MemoryArrayType (0 to MemoryPtr.NoOfWords - 1);
  FOR i IN 0 TO MemoryPtr.NoOfWords - 1 LOOP
    MemoryPtr.MemoryArrayPtr(i)
      := new MemoryWordType (MemoryPtr.NoOfBitsPerWord - 1 DOWNTO 0);
  END LOOP;
  IF (MemoryLoadFile /= "") THEN
    LoadMemory (MemoryPtr, MemoryLoadFile, BinaryLoadFile);
  END IF;
  RETURN MemoryPtr;
END;

-- ------------------------------------------------------------------------
-- Procedure:   VitalMemoryTable
-- Parameters:  DataOutBus     - Output candidate zero delay data bus out
--              MemoryData     - Pointer to memory data structure
--              PrevControls   - Previous data in for edge detection
--              PrevEnableBus  - Previous enables for edge detection
--              PrevDataInBus  - Previous data bus for edge detection
--              PrevAddressBus - Previous address bus for edge detection
--              PortFlag       - Indicates port operating mode
--              PortFlagArray  - Vector form of PortFlag for sub-word
--              Controls       - Agregate of scalar control lines
--              EnableBus      - Concatenation of vector control lines
--              DataInBus      - Input value of data bus in
```

```
--              AddressBus     - Input value of address bus in
--              AddressValue   - Decoded value of the AddressBus
--              MemoryTable    - Input memory action table
--              PortType       - The type of port (currently not used)
--              PortName       - Port name string for messages
--              HeaderMsg      - Header string for messages
--              MsgOn          - Control the generation of messages
--              MsgSeverity    - Control level of message generation
-- Description: This procedure implements the majority of the memory
--              modeling functionality via lookup of the memory action
--              tables and performing the specified actions if matches
--              are found, or the default actions otherwise.  The
--              overloadings are provided for the word and sub-word
--              (using the EnableBus and PortFlagArray arguments) addressing
--              cases.
-- ----------------------------------------------------------------------------
PROCEDURE VitalMemoryTable (
  VARIABLE DataOutBus     : INOUT std_logic_vector;
  VARIABLE MemoryData     : INOUT VitalMemoryDataType;
  VARIABLE PrevControls   : INOUT std_logic_vector;
  VARIABLE PrevDataInBus  : INOUT std_logic_vector;
  VARIABLE PrevAddressBus : INOUT std_logic_vector;
  VARIABLE PortFlag       : INOUT VitalPortFlagVectorType;
  CONSTANT Controls       : IN std_logic_vector;
  CONSTANT DataInBus      : IN std_logic_vector;
  CONSTANT AddressBus     : IN std_logic_vector;
  VARIABLE AddressValue   : INOUT VitalAddressValueType;
  CONSTANT MemoryTable    : IN VitalMemoryTableType;
  CONSTANT PortType       : IN VitalPortType := UNDEF;
  CONSTANT PortName       : IN STRING := "";
  CONSTANT HeaderMsg      : IN STRING := "";
  CONSTANT MsgOn          : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
) IS

  VARIABLE DataOutTmp     : std_logic_vector(DataOutBus'RANGE)
                            := DataOutBus;
  VARIABLE MemoryAction   : VitalMemorySymbolType;
  VARIABLE DataAction     : VitalMemorySymbolType;
  VARIABLE HighBit        : NATURAL := MemoryData.NoOfBitsPerWord-1;
  VARIABLE LowBit         : NATURAL := 0;
  VARIABLE Address        : INTEGER := 0;
  VARIABLE PortFlagTmp    : VitalPortFlagType;
  VARIABLE AddrFlag       : VitalMemorySymbolType := 'g';  -- good addr
  VARIABLE DataFlag       : VitalMemorySymbolType := 'g';  -- good data
  VARIABLE MemCorruptMask : std_logic_vector (DataOutBus'RANGE);
  VARIABLE DatCorruptMask : std_logic_vector (DataOutBus'RANGE);

BEGIN

  -- Optimize for case when all current inputs are same as previous
  IF (PrevDataInBus = DataInBus
      AND PrevAddressBus = AddressBus
      AND PrevControls = Controls
      AND PortFlag(0).MemoryCurrent = PortFlag(0).MemoryPrevious
      AND PortFlag(0).DataCurrent = PortFlag(0).DataPrevious) THEN
    PortFlag(0).OutputDisable := TRUE;
    RETURN;
  END IF;

  PortFlag(0).DataPrevious := PortFlag(0).DataCurrent;
  PortFlag(0).MemoryPrevious := PortFlag(0).MemoryCurrent;
  PortFlag(0).OutputDisable := FALSE;
  PortFlagTmp := PortFlag(0);

  -- Convert address bus to integer value and table lookup flag
  DecodeAddress(
    Address         => Address            ,
    AddrFlag        => AddrFlag           ,
    MemoryData      => MemoryData         ,
    PrevAddressBus  => PrevAddressBus     ,
    AddressBus      => AddressBus
  );

  -- Interpret data bus as a table lookup flag
  DecodeData (
    DataFlag        => DataFlag          ,
    PrevDataInBus   => PrevDataInBus     ,
    DataInBus       => DataInBus         ,
    HighBit         => HighBit           ,
    LowBit          => LowBit
```

 Copyright © 2001 IEEE. All rights reserved.

```
    );

    -- Lookup memory and data actions
    MemoryTableLookUp(
      MemoryAction        => MemoryAction      ,
      DataAction          => DataAction        ,
      MemoryCorruptMask   => MemCorruptMask    ,
      DataCorruptMask     => DatCorruptMask    ,
      PrevControls        => PrevControls      ,
      Controls            => Controls          ,
      AddrFlag            => AddrFlag          ,
      DataFlag            => DataFlag          ,
      MemoryTable         => MemoryTable       ,
      PortName            => PortName          ,
      HeaderMsg           => HeaderMsg         ,
      MsgOn               => MsgOn
    );

    -- Handle data action before memory action
    -- This allows reading previous memory contents
    HandleDataAction(
      DataOutBus      => DataOutTmp       ,
      MemoryData      => MemoryData       ,
      PortFlag        => PortFlagTmp      ,
      CorruptMask     => DatCorruptMask   ,
      DataInBus       => DataInBus        ,
      Address         => Address          ,
      HighBit         => HighBit          ,
      LowBit          => LowBit           ,
      MemoryTable     => MemoryTable      ,
      DataAction      => DataAction       ,
      CallerName      => MsgVMT           ,
      PortName        => PortName         ,
      HeaderMsg       => HeaderMsg        ,
      MsgOn           => MsgOn
    );

    HandleMemoryAction(
      MemoryData      => MemoryData       ,
      PortFlag        => PortFlagTmp      ,
      CorruptMask     => MemCorruptMask   ,
      DataInBus       => DataInBus        ,
      Address         => Address          ,
      HighBit         => HighBit          ,
      LowBit          => LowBit           ,
      MemoryTable     => MemoryTable      ,
      MemoryAction    => MemoryAction     ,
      CallerName      => MsgVMT           ,
      PortName        => PortName         ,
      HeaderMsg       => HeaderMsg        ,
      MsgOn           => MsgOn
    );

    -- Set the output PortFlag(0) value
    IF (DataAction = 'S') THEN
      PortFlagTmp.OutputDisable := TRUE;
    END IF;
    IF (PortFlagTmp.DataCurrent = PortFlagTmp.DataPrevious
        AND PortFlagTmp.DataCurrent = HIGHZ) THEN
      PortFlagTmp.OutputDisable := TRUE;
    END IF;
    PortFlag(0) := PortFlagTmp;

    -- Set previous values for subsequent edge detection
    PrevControls := Controls;
    PrevDataInBus := DataInBus;
    PrevAddressBus := AddressBus;

    -- Set the candidate zero delay return value
    DataOutBus := DataOutTmp;

    -- Set the output AddressValue for VitalMemoryCrossPorts
    AddressValue := Address;

  END VitalMemoryTable;

  -- ----------------------------------------------------------------------------
  PROCEDURE VitalMemoryTable (
    VARIABLE DataOutBus     : INOUT std_logic_vector;
    VARIABLE MemoryData     : INOUT VitalMemoryDataType;
    VARIABLE PrevControls   : INOUT std_logic_vector;
```


Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE PrevEnableBus  : INOUT std_logic_vector;
    VARIABLE PrevDataInBus  : INOUT std_logic_vector;
    VARIABLE PrevAddressBus : INOUT std_logic_vector;
    VARIABLE PortFlagArray  : INOUT VitalPortFlagVectorType;
    CONSTANT Controls       : IN std_logic_vector;
    CONSTANT EnableBus      : IN std_logic_vector;
    CONSTANT DataInBus      : IN std_logic_vector;
    CONSTANT AddressBus     : IN std_logic_vector;
    VARIABLE AddressValue   : INOUT VitalAddressValueType;
    CONSTANT MemoryTable    : IN VitalMemoryTableType;
    CONSTANT PortType       : IN VitalPortType := UNDEF;
    CONSTANT PortName       : IN STRING := "";
    CONSTANT HeaderMsg      : IN STRING := "";
    CONSTANT MsgOn          : IN BOOLEAN := TRUE;
    CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING
  ) IS

    VARIABLE BitsPerWord    : NATURAL := MemoryData.NoOfBitsPerWord;
    VARIABLE BitsPerSubWord : NATURAL := MemoryData.NoOfBitsPerSubWord;
    VARIABLE BitsPerEnable  : NATURAL := MemoryData.NoOfBitsPerEnable;
    VARIABLE DataOutTmp     : std_logic_vector(DataOutBus'RANGE)
    := DataOutBus;
    VARIABLE MemoryAction   : VitalMemorySymbolType;
    VARIABLE DataAction     : VitalMemorySymbolType;
    VARIABLE HighBit        : NATURAL := BitsPerSubWord-1;
    VARIABLE LowBit         : NATURAL := 0;
    VARIABLE Address        : INTEGER := 0;
    VARIABLE PortFlagTmp    : VitalPortFlagType;
    VARIABLE AddrFlag       : VitalMemorySymbolType := 'g';  -- good addr
    VARIABLE DataFlag       : VitalMemorySymbolType := 'g';  -- good data
    VARIABLE MemCorruptMask : std_logic_vector (DataOutBus'RANGE);
    VARIABLE DatCorruptMask : std_logic_vector (DataOutBus'RANGE);

  BEGIN

    -- Optimize for case when all current inputs are same as previous
    IF (PrevDataInBus = DataInBus
        AND PrevAddressBus = AddressBus
        AND PrevControls = Controls) THEN
      CheckFlags:
      FOR i IN 0 TO BitsPerEnable-1 LOOP
        IF (PortFlagArray(i).MemoryCurrent /= PortFlagArray(i).MemoryPrevious
            OR  PortFlagArray(i).DataCurrent /= PortFlagArray(i).DataPrevious) THEN
          EXIT CheckFlags;
        END IF;
        IF (i = BitsPerEnable-1) THEN
          FOR j IN 0 TO BitsPerEnable-1 LOOP
            PortFlagArray(j).OutputDisable := TRUE;
          END LOOP;
          RETURN;
        END IF;
      END LOOP;
    END IF;

    -- Convert address bus to integer value and table lookup flag
    DecodeAddress(
      Address          => Address,
      AddrFlag         => AddrFlag,
      MemoryData       => MemoryData,
      PrevAddressBus   => PrevAddressBus,
      AddressBus       => AddressBus
    );

    -- Perform independent operations for each sub-word
    FOR i IN 0 TO BitsPerEnable-1 LOOP

      -- Set the output PortFlag(i) value
      PortFlagArray(i).DataPrevious := PortFlagArray(i).DataCurrent;
      PortFlagArray(i).MemoryPrevious := PortFlagArray(i).MemoryCurrent;
      PortFlagArray(i).OutputDisable := FALSE;
      PortFlagTmp := PortFlagArray(i);

      -- Interpret data bus as a table lookup flag
      DecodeData (
        DataFlag        => DataFlag         ,
        PrevDataInBus   => PrevDataInBus    ,
        DataInBus       => DataInBus        ,
        HighBit         => HighBit          ,
        LowBit          => LowBit
      );
```

 Copyright © 2001 IEEE. All rights reserved.

```
-- Lookup memory and data actions
MemoryTableLookUp(
  MemoryAction        => MemoryAction     ,
  DataAction          => DataAction       ,
  MemoryCorruptMask   => MemCorruptMask   ,
  DataCorruptMask     => DatCorruptMask   ,
  PrevControls        => PrevControls     ,
  PrevEnableBus       => PrevEnableBus    ,
  Controls            => Controls         ,
  EnableBus           => EnableBus        ,
  EnableIndex         => i                ,
  BitsPerWord         => BitsPerWord      ,
  BitsPerSubWord      => BitsPerSubWord   ,
  BitsPerEnable       => BitsPerEnable    ,
  AddrFlag            => AddrFlag         ,
  DataFlag            => DataFlag         ,
  MemoryTable         => MemoryTable      ,
  PortName            => PortName         ,
  HeaderMsg           => HeaderMsg        ,
  MsgOn               => MsgOn
);

-- Handle data action before memory action
-- This allows reading previous memory contents
HandleDataAction(
  DataOutBus      => DataOutTmp       ,
  MemoryData      => MemoryData       ,
  PortFlag        => PortFlagTmp      ,
  CorruptMask     => DatCorruptMask   ,
  DataInBus       => DataInBus        ,
  Address         => Address          ,
  HighBit         => HighBit          ,
  LowBit          => LowBit           ,
  MemoryTable     => MemoryTable      ,
  DataAction      => DataAction       ,
  CallerName      => MsgVMT           ,
  PortName        => PortName         ,
  HeaderMsg       => HeaderMsg        ,
  MsgOn           => MsgOn
);

HandleMemoryAction(
  MemoryData      => MemoryData       ,
  PortFlag        => PortFlagTmp      ,
  CorruptMask     => MemCorruptMask   ,
  DataInBus       => DataInBus        ,
  Address         => Address          ,
  HighBit         => HighBit          ,
  LowBit          => LowBit           ,
  MemoryTable     => MemoryTable      ,
  MemoryAction    => MemoryAction     ,
  CallerName      => MsgVMT           ,
  PortName        => PortName         ,
  HeaderMsg       => HeaderMsg        ,
  MsgOn           => MsgOn
);

-- Set the output PortFlag(i) value
IF (DataAction = 'S') THEN
  PortFlagTmp.OutputDisable := TRUE;
END IF;
IF (PortFlagTmp.DataCurrent = PortFlagTmp.DataPrevious
    AND PortFlagTmp.DataCurrent = HIGHZ) THEN
  PortFlagTmp.OutputDisable := TRUE;
END IF;
PortFlagArray(i) := PortFlagTmp;

IF (i < BitsPerEnable-1) THEN
  -- Calculate HighBit and LowBit
  LowBit := LowBit + BitsPerSubWord;
  IF (LowBit > BitsPerWord) THEN
    LowBit := BitsPerWord;
  END IF;
  HighBit := LowBit + BitsPerSubWord;
  IF (HighBit > BitsPerWord) THEN
    HighBit := BitsPerWord;
  ELSE
    HighBit := HighBit - 1;
  END IF;
END IF;
```

Copyright © 2001 IEEE. All rights reserved.

```
  END LOOP;

  -- Set previous values for subsequent edge detection
  PrevControls := Controls;
  PrevEnableBus := EnableBus;
  PrevDataInBus := DataInBus;
  PrevAddressBus := AddressBus;

  -- Set the candidate zero delay return value
  DataOutBus := DataOutTmp;

  -- Set the output AddressValue for VitalMemoryCrossPorts
  AddressValue := Address;

END VitalMemoryTable;

-- ----------------------------------------------------------------------------
-- Procedure:   VitalMemoryCrossPorts
-- Parameters:  DataOutBus     - Output candidate zero delay data bus out
--              MemoryData     - Pointer to memory data structure
--              SamePortFlag   - Operating mode for same port
--              SamePortAddressValue  - Operating modes for cross ports
--              CrossPortAddressArray - Decoded AddressBus for cross ports
--              CrossPortMode  - Write contention and crossport read control
--              PortName       - Port name string for messages
--              HeaderMsg      - Header string for messages
--              MsgOn          - Control the generation of messages
-- Description: These procedures control the effect of memory operations
--              on a given port due to operations on other ports in a
--              multi-port memory.
--              This includes data write through when reading and writing
--              to the same address, as well as write contention when
--              there are multiple write to the same address.
--              If addresses do not match then data bus is unchanged.
--              The DataOutBus can be diabled with 'Z' value.
--              If the WritePortFlag is 'CORRUPT', that would mean
--              that the whole memory is corrupted. So, for corrupting
--              the Read port, the Addresses need not be compared.
--
--     CrossPortMode Enum            Description
-- 1.  CpRead                        Allows Cross Port Read Only
--                                   No contention checking.
-- 2.  WriteContention               Allows for write contention checks
--                                   only between multiple write ports
-- 3.  ReadWriteContention           Allows contention between read and
--                                   write ports. The action is to corrupt
--                                   the memory and the output bus.
-- 4.  CpReadAndWriteContention      Is a combination of 1 & 2
-- 5.  CpReadAndReadContention       Allows contention between read and
--                                   write ports. The action is to corrupt
--                                   the dataout bus only. The cp read is
--                                   performed if not contending.
-- ----------------------------------------------------------------------------
PROCEDURE VitalMemoryCrossPorts (
  VARIABLE DataOutBus             : INOUT std_logic_vector;
  VARIABLE MemoryData             : INOUT VitalMemoryDataType;
  VARIABLE SamePortFlag           : INOUT VitalPortFlagVectorType;
  CONSTANT SamePortAddressValue   : IN VitalAddressValueType;
  CONSTANT CrossPortFlagArray     : IN VitalPortFlagVectorType;
  CONSTANT CrossPortAddressArray  : IN VitalAddressValueVectorType;
  CONSTANT CrossPortMode          : IN VitalCrossPortModeType
                                        := CpReadAndWriteContention;
  CONSTANT PortName               : IN STRING := "";
  CONSTANT HeaderMsg              : IN STRING := "";
  CONSTANT MsgOn                  : IN BOOLEAN := TRUE
) IS

  VARIABLE BitsPerWord    : NATURAL := MemoryData.NoOfBitsPerWord;
  VARIABLE BitsPerSubWord : NATURAL := MemoryData.NoOfBitsPerSubWord;
  VARIABLE BitsPerEnable  : NATURAL := MemoryData.NoOfBitsPerEnable;
  VARIABLE DataOutTmp : std_logic_vector(DataOutBus'RANGE) := (OTHERS => 'Z');
  VARIABLE MemoryTmp  : std_logic_vector(DataOutBus'RANGE);
  VARIABLE CrossPorts : NATURAL := CrossPortAddressArray'LENGTH;
  VARIABLE LowBit     : NATURAL := 0;
  VARIABLE HighBit    : NATURAL := BitsPerSubWord-1;
  VARIABLE Address    : VitalAddressValueType := SamePortAddressValue;
  VARIABLE AddressJ   : VitalAddressValueType;
  VARIABLE AddressK   : VitalAddressValueType;
  VARIABLE PortFlagI  : VitalPortFlagType;
  VARIABLE PortFlagIJ : VitalPortFlagType;
  VARIABLE PortFlagIK : VitalPortFlagType;
```

 Copyright © 2001 IEEE. All rights reserved.

```
    VARIABLE DoCpRead   : BOOLEAN := FALSE;
    VARIABLE DoWrCont   : BOOLEAN := FALSE;
    VARIABLE DoCpCont   : BOOLEAN := FALSE;
    VARIABLE DoRdWrCont : BOOLEAN := FALSE;
    VARIABLE CpWrCont   : BOOLEAN := FALSE;
    VARIABLE ModeWrCont : BOOLEAN :=
                            (CrossPortMode=WriteContention) OR
                            (CrossPortMode=CpReadAndWriteContention);
    VARIABLE ModeCpRead : BOOLEAN :=
                            (CrossPortMode=CpRead) OR
                            (CrossPortMode=CpReadAndWriteContention);
    VARIABLE ModeCpCont : BOOLEAN := (CrossPortMode=ReadWriteContention);
    VARIABLE ModeRdWrCont : BOOLEAN := (CrossPortMode=CpReadAndReadContention);

  BEGIN

    -- Check for disabled port (i.e. OTHERS => 'Z')
    IF (DataOutBus = DataOutTmp) THEN
      RETURN;
    ELSE
      DataOutTmp := DataOutBus;
    END IF;

    -- Check for error in address
    IF (Address < 0) THEN
      RETURN;
    END IF;

    ReadMemory(MemoryData,MemoryTmp,Address);

    SubWordLoop: -- For each slice of the sub-word I
    FOR i IN 0 TO BitsPerEnable-1 LOOP
      PortFlagI := SamePortFlag(i);

      -- For each cross port J: check with same port address
      FOR j IN 0 TO CrossPorts-1 LOOP
        PortFlagIJ := CrossPortFlagArray(i+j*BitsPerEnable);
        AddressJ := CrossPortAddressArray(j);
        IF (AddressJ < 0) THEN
          NEXT;
        END IF;
        DoWrCont :=  (Address    = AddressJ)  AND
                     (ModeWrCont = TRUE)      AND
                    ((PortFlagI.MemoryCurrent  = WRITE)     OR
                     (PortFlagI.MemoryCurrent  = CORRUPT))  AND
                    ((PortFlagIJ.MemoryCurrent = WRITE)     OR
                     (PortFlagIJ.MemoryCurrent = CORRUPT))  ;
        DoCpRead :=  (Address    = AddressJ)  AND
                     (ModeCpRead = TRUE)      AND
                    ((PortFlagI.MemoryCurrent  = READ)      OR
                     (PortFlagI.OutputDisable  = TRUE))     AND
                    ((PortFlagIJ.MemoryCurrent = WRITE)     OR
                     (PortFlagIJ.MemoryCurrent = CORRUPT))  ;
        DoCpCont :=  (Address    = AddressJ)  AND
                     (ModeCpCont = TRUE)      AND
                    ((PortFlagI.MemoryCurrent  = READ)      OR
                     (PortFlagI.OutputDisable  = TRUE))     AND
                    ((PortFlagIJ.MemoryCurrent = WRITE)     OR
                     (PortFlagIJ.MemoryCurrent = CORRUPT))  ;
        DoRdWrCont:= (Address    = AddressJ)  AND
                     (ModeRdWrCont = TRUE)    AND
                    ((PortFlagI.MemoryCurrent  = READ)      OR
                     (PortFlagI.OutputDisable  = TRUE))     AND
                    ((PortFlagIJ.MemoryCurrent = WRITE)     OR
                     (PortFlagIJ.MemoryCurrent = CORRUPT))  ;
        IF (DoWrCont OR DoCpCont) THEN
          -- Corrupt dataout and memory
          MemoryTmp(HighBit DOWNTO LowBit) := (OTHERS => 'X');
          DataOutTmp(HighBit DOWNTO LowBit) := (OTHERS => 'X');
          SamePortFlag(i).MemoryCurrent := CORRUPT;
          SamePortFlag(i).DataCurrent := CORRUPT;
          SamePortFlag(i).OutputDisable := FALSE;
          EXIT;
        END IF;
        IF (DoCpRead) THEN
          -- Update dataout with memory
          DataOutTmp(HighBit DOWNTO LowBit) :=
          MemoryTmp(HighBit DOWNTO LowBit);
          SamePortFlag(i).MemoryCurrent := READ;
          SamePortFlag(i).DataCurrent := READ;
          SamePortFlag(i).OutputDisable := FALSE;
```

Copyright © 2001 IEEE. All rights reserved.

```
        EXIT;
      END IF;
      IF (DoRdWrCont) THEN
        -- Corrupt dataout only
        DataOutTmp(HighBit DOWNTO LowBit) := (OTHERS => 'X');
        SamePortFlag(i).DataCurrent := CORRUPT;
        SamePortFlag(i).OutputDisable := FALSE;
        EXIT;
      END IF;
    END LOOP;

    IF (i < BitsPerEnable-1) THEN
      -- Calculate HighBit and LowBit
      LowBit := LowBit + BitsPerSubWord;
      IF (LowBit > BitsPerWord) THEN
        LowBit := BitsPerWord;
      END IF;
      HighBit := LowBit + BitsPerSubWord;
      IF (HighBit > BitsPerWord) THEN
        HighBit := BitsPerWord;
      ELSE
        HighBit := HighBit - 1;
      END IF;
    END IF;

  END LOOP;   -- SubWordLoop

  DataOutBus := DataOutTmp;

  IF (DoWrCont) THEN
    IF (MsgOn) THEN
      PrintMemoryMessage(MsgVMCP,ErrMcpWrCont,HeaderMsg,PortName);
    END IF;
    WriteMemory(MemoryData,MemoryTmp,Address);
  END IF;

  IF (DoCpCont) THEN
    IF (MsgOn) THEN
      PrintMemoryMessage(MsgVMCP,ErrMcpCpCont,HeaderMsg,PortName);
    END IF;
    WriteMemory(MemoryData,MemoryTmp,Address);
  END IF;

  IF (DoCpRead) THEN
    IF (MsgOn) THEN
      PrintMemoryMessage(MsgVMCP,ErrMcpCpRead,HeaderMsg,PortName);
    END IF;
  END IF;

  IF (DoRdWrCont) THEN
    IF (MsgOn) THEN
      PrintMemoryMessage(MsgVMCP,ErrMcpRdWrCo,HeaderMsg,PortName);
    END IF;
  END IF;

END VitalMemoryCrossPorts;

-- ----------------------------------------------------------------------------
PROCEDURE VitalMemoryCrossPorts (
  VARIABLE MemoryData             : INOUT VitalMemoryDataType;
  CONSTANT CrossPortFlagArray     : IN VitalPortFlagVectorType;
  CONSTANT CrossPortAddressArray  : IN VitalAddressValueVectorType;
  CONSTANT HeaderMsg              : IN STRING := "";
  CONSTANT MsgOn                  : IN BOOLEAN := TRUE
) IS

  VARIABLE BitsPerWord    : NATURAL := MemoryData.NoOfBitsPerWord;
  VARIABLE BitsPerSubWord : NATURAL := MemoryData.NoOfBitsPerSubWord;
  VARIABLE BitsPerEnable  : NATURAL := MemoryData.NoOfBitsPerEnable;
  VARIABLE MemoryTmp  : std_logic_vector(BitsPerWord-1 DOWNTO 0);
  VARIABLE CrossPorts : NATURAL := CrossPortAddressArray'LENGTH;
  VARIABLE LowBit     : NATURAL := 0;
  VARIABLE HighBit    : NATURAL := BitsPerSubWord-1;
  VARIABLE AddressJ   : VitalAddressValueType;
  VARIABLE AddressK   : VitalAddressValueType;
  VARIABLE PortFlagIJ : VitalPortFlagType;
  VARIABLE PortFlagIK : VitalPortFlagType;
  VARIABLE CpWrCont   : BOOLEAN := FALSE;

BEGIN
```

 Copyright © 2001 IEEE. All rights reserved.

```
    SubWordLoop: -- For each slice of the sub-word I
    FOR i IN 0 TO BitsPerEnable-1 LOOP

      -- For each cross port J: check with each cross port K
      FOR j IN 0 TO CrossPorts-1 LOOP
        PortFlagIJ := CrossPortFlagArray(i+j*BitsPerEnable);
        AddressJ := CrossPortAddressArray(j);
        -- Check for error in address
        IF (AddressJ < 0) THEN
          NEXT;
        END IF;
        ReadMemory(MemoryData,MemoryTmp,AddressJ);
        -- For each cross port K
        FOR k IN 0 TO CrossPorts-1 LOOP
          IF (k <= j) THEN
            NEXT;
          END IF;
          PortFlagIK := CrossPortFlagArray(i+k*BitsPerEnable);
          AddressK := CrossPortAddressArray(k);
          -- Check for error in address
          IF (AddressK < 0) THEN
            NEXT;
          END IF;
          CpWrCont := ( (AddressJ   = AddressK) AND
                        (PortFlagIJ.MemoryCurrent = WRITE)    AND
                        (PortFlagIK.MemoryCurrent = WRITE) ) OR
                      ( (PortFlagIJ.MemoryCurrent = WRITE)    AND
                        (PortFlagIK.MemoryCurrent = CORRUPT) ) OR
                      ( (PortFlagIJ.MemoryCurrent = CORRUPT) AND
                        (PortFlagIK.MemoryCurrent = WRITE) ) OR
                      ( (PortFlagIJ.MemoryCurrent = CORRUPT) AND
                        (PortFlagIK.MemoryCurrent = CORRUPT) ) ;
          IF (CpWrCont) THEN
            -- Corrupt memory only
            MemoryTmp(HighBit DOWNTO LowBit) := (OTHERS => 'X');
            EXIT;
          END IF;
        END LOOP;   -- FOR k IN 0 TO CrossPorts-1 LOOP
        IF (CpWrCont = TRUE) THEN
          IF (MsgOn) THEN
            PrintMemoryMessage(MsgVMCP,ErrMcpCpWrCont,HeaderMsg);
          END IF;
          WriteMemory(MemoryData,MemoryTmp,AddressJ);
        END IF;
      END LOOP;   -- FOR j IN 0 TO CrossPorts-1 LOOP

      IF (i < BitsPerEnable-1) THEN
        -- Calculate HighBit and LowBit
        LowBit := LowBit + BitsPerSubWord;
        IF (LowBit > BitsPerWord) THEN
          LowBit := BitsPerWord;
        END IF;
        HighBit := LowBit + BitsPerSubWord;
        IF (HighBit > BitsPerWord) THEN
          HighBit := BitsPerWord;
        ELSE
          HighBit := HighBit - 1;
        END IF;
      END IF;
    END LOOP;   -- SubWordLoop

  END VitalMemoryCrossPorts;


  -- ------------------------------------------------------------------------------
  -- Procedure:   VitalMemoryViolation
  -- Parameters:  DataOutBus        - Output zero delay data bus out
  --              MemoryData        - Pointer to memory data structure
  --              PortFlag          - Indicates port operating mode
  --              TimingDataArray   - This is currently not used (comment out)
  --              ViolationArray    - Aggregation of violation variables
  --              DataInBus         - Input value of data bus in
  --              AddressBus        - Input value of address bus in
  --              AddressValue      - Decoded value of the AddressBus
  --              ViolationTable    - Input memory violation table
  --              PortName          - Port name string for messages
  --              HeaderMsg         - Header string for messages
  --              MsgOn             - Control the generation of messages
  --              MsgSeverity       - Control level of message generation
  -- Description: This procedure is intended to implement all actions on the
  --              memory contents and data out bus as a result of timing viols.
```

Copyright © 2001 IEEE. All rights reserved.

```
--              It uses the memory action table to perform various corruption
--              policies specified by the user.
-- ----------------------------------------------------------------------------

PROCEDURE VitalMemoryViolation (
  VARIABLE DataOutBus           : INOUT std_logic_vector;
  VARIABLE MemoryData           : INOUT VitalMemoryDataType;
  VARIABLE PortFlag             : INOUT VitalPortFlagVectorType;
  CONSTANT DataInBus            : IN std_logic_vector;
  CONSTANT AddressValue         : IN VitalAddressValueType;
  CONSTANT ViolationFlags       : IN std_logic_vector;
  CONSTANT ViolationFlagsArray  : IN X01ArrayT;
  CONSTANT ViolationSizesArray  : IN VitalMemoryViolFlagSizeType;
  CONSTANT ViolationTable       : IN VitalMemoryTableType;
  CONSTANT PortType             : IN VitalPortType;
  CONSTANT PortName             : IN STRING := "";
  CONSTANT HeaderMsg            : IN STRING := "";
  CONSTANT MsgOn                : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity          : IN SEVERITY_LEVEL := WARNING
) IS

  VARIABLE BitsPerWord    : NATURAL := MemoryData.NoOfBitsPerWord;
  VARIABLE BitsPerSubWord : NATURAL := MemoryData.NoOfBitsPerSubWord;
  VARIABLE BitsPerEnable  : NATURAL := MemoryData.NoOfBitsPerEnable;
  VARIABLE DataOutTmp     : std_logic_vector(DataOutBus'RANGE)
                            := DataOutBus;
  VARIABLE MemoryAction   : VitalMemorySymbolType;
  VARIABLE DataAction     : VitalMemorySymbolType;
  -- VMT relies on the corrupt masks so HighBit/LowBit are full word
  VARIABLE HighBit        : NATURAL := BitsPerWord-1;
  VARIABLE LowBit         : NATURAL := 0;
  VARIABLE PortFlagTmp    : VitalPortFlagType;
  VARIABLE VFlagArrayTmp  : std_logic_vector
                            (0 TO ViolationFlagsArray'LENGTH-1);
  VARIABLE MemCorruptMask : std_logic_vector (DataOutBus'RANGE);
  VARIABLE DatCorruptMask : std_logic_vector (DataOutBus'RANGE);

BEGIN

  -- Don't do anything if given an error address
  IF (AddressValue < 0) THEN
    RETURN;
  END IF;

  FOR i IN ViolationFlagsArray'RANGE LOOP
    VFlagArrayTmp(i) := ViolationFlagsArray(i);
  END LOOP;

  -- Lookup memory and data actions
  ViolationTableLookUp(
    MemoryAction        => MemoryAction         ,
    DataAction          => DataAction           ,
    MemoryCorruptMask   => MemCorruptMask       ,
    DataCorruptMask     => DatCorruptMask       ,
    ViolationFlags      => ViolationFlags       ,
    ViolationFlagsArray => VFlagArrayTmp        ,
    ViolationSizesArray => ViolationSizesArray  ,
    ViolationTable      => ViolationTable       ,
    BitsPerWord         => BitsPerWord          ,
    BitsPerSubWord      => BitsPerSubWord       ,
    BitsPerEnable       => BitsPerEnable        ,
    PortName            => PortName             ,
    HeaderMsg           => HeaderMsg            ,
    MsgOn               => MsgOn
  );

  -- Need to read incoming PF value (was not before)
  PortFlagTmp := PortFlag(0);

  IF (PortType = READ OR PortType = RDNWR) THEN
    -- Handle data action before memory action
    -- This allows reading previous memory contents
    HandleDataAction(
      DataOutBus      => DataOutTmp       ,
      MemoryData      => MemoryData       ,
      PortFlag        => PortFlagTmp      ,
      CorruptMask     => DatCorruptMask   ,
      DataInBus       => DataInBus        ,
      Address         => AddressValue     ,
      HighBit         => HighBit          ,
      LowBit          => LowBit           ,
```

 Copyright © 2001 IEEE. All rights reserved.

```
        MemoryTable     => ViolationTable   ,
        DataAction      => DataAction       ,
        CallerName      => MsgVMV           ,
        PortName        => PortName         ,
        HeaderMsg       => HeaderMsg        ,
        MsgOn           => MsgOn
      );
    END IF;

    IF (PortType = WRITE OR PortType = RDNWR) THEN
      HandleMemoryAction(
        MemoryData      => MemoryData       ,
        PortFlag        => PortFlagTmp      ,
        CorruptMask     => MemCorruptMask   ,
        DataInBus       => DataInBus        ,
        Address         => AddressValue     ,
        HighBit         => HighBit          ,
        LowBit          => LowBit           ,
        MemoryTable     => ViolationTable   ,
        MemoryAction    => MemoryAction     ,
        CallerName      => MsgVMV           ,
        PortName        => PortName         ,
        HeaderMsg       => HeaderMsg        ,
        MsgOn           => MsgOn
      );
    END IF;

    -- Check if we need to turn off PF.OutputDisable
    IF (DataAction /= 'S') THEN
      PortFlagTmp.OutputDisable := FALSE;
      -- Set the output PortFlag(0) value
      -- Note that all bits of PortFlag get PortFlagTmp
      FOR i IN PortFlag'RANGE LOOP
        PortFlag(i) := PortFlagTmp;
      END LOOP;
    END IF;

    -- Set the candidate zero delay return value
    DataOutBus := DataOutTmp;

  END;

  PROCEDURE VitalMemoryViolation (
    VARIABLE DataOutBus           : INOUT std_logic_vector;
    VARIABLE MemoryData           : INOUT VitalMemoryDataType;
    VARIABLE PortFlag             : INOUT VitalPortFlagVectorType;
    CONSTANT DataInBus            : IN std_logic_vector;
    CONSTANT AddressValue         : IN VitalAddressValueType;
    CONSTANT ViolationFlags       : IN std_logic_vector;
    CONSTANT ViolationTable       : IN VitalMemoryTableType;
    CONSTANT PortType             : IN VitalPortType;
    CONSTANT PortName             : IN STRING := "";
    CONSTANT HeaderMsg            : IN STRING := "";
    CONSTANT MsgOn                : IN BOOLEAN := TRUE;
    CONSTANT MsgSeverity          : IN SEVERITY_LEVEL := WARNING
  ) IS

    VARIABLE VFlagArrayTmp  : X01ArrayT (0 TO 0);

  BEGIN

    VitalMemoryViolation (
      DataOutBus              => DataOutBus       ,
      MemoryData              => MemoryData       ,
      PortFlag                => PortFlag         ,
      DataInBus               => DataInBus        ,
      AddressValue            => AddressValue     ,
      ViolationFlags          => ViolationFlags   ,
      ViolationFlagsArray     => VFlagArrayTmp    ,
      ViolationSizesArray     => ( 0 => 0 )       ,
      ViolationTable          => ViolationTable   ,
      PortType                => PortType         ,
      PortName                => PortName         ,
      HeaderMsg               => HeaderMsg        ,
      MsgOn                   => MsgOn            ,
      MsgSeverity             => MsgSeverity
    );

  END;

  END Vital_Memory ;
```

Copyright © 2001 IEEE. All rights reserved.

Annex A

(informative)

Syntax summary

VITAL_control_generic_declaration ::= [p. 16]
[**constant**] identifier_list ::= [**in**] type_mark [index_constraint] [:= *static*_expression] ;

VITAL_design_file ::= [p. 8]
VITAL_design_unit { VITAL_design_unit }

VITAL_design_unit ::= [p. 8]
context_clause library_unit
| context_clause VITAL_library_unit

VITAL_entity_declarative_part ::= VITAL_Level0_attribute_specification [p. 8]

VITAL_entity_generic_clause ::= [p. 8]
generic (VITAL_entity_interface_list) ;

VITAL_entity_header ::= [p. 8]
[VITAL_entity_generic_clause]
[VITAL_entity_port_clause]

VITAL_entity_interface_declaration ::= [p. 8]
interface_constant_declaration
| VITAL_timing_generic_declaration
| VITAL_control_generic_declaration
| VITAL_entity_port_declaration

VITAL_entity_interface_list ::= [p. 8]
VITAL_entity_interface_declaration { ; VITAL_entity_interface_declaration }

VITAL_entity_port_clause ::= [p. 8]
port (VITAL_entity_interface_list) ;

VITAL_entity_port_declaration ::= [p. 9]
[**signal**] identifier_list : [mode] type_mark [index_constraint] [:= *static*_expression] ;

VITAL_functionality_section ::= [p. 43]
{ VITAL_variable_assignment_statement | procedure_call_statement }

VITAL_internal_signal_declaration ::= [p. 36]
signal identifier_list : type_mark [index_constraint] [:= expression] ;

 Copyright © 2001 IEEE. All rights reserved.

VITAL_Level_0_architecture_body ::= [p. 17]
architecture identifier **of** *entity*_name **is**
VITAL_Level_0_ architecture_declarative_part
begin
architecture_statement_part
end [architecture] [*architecture*_simple_name] ;

VITAL_Level_0_architecture_declarative_part ::= [p. 17]
VITAL_Level0_attribute_specification { block_declarative_item }

VITAL_Level_0_entity_declaration ::= [p. 8]
entity identifier **is**
VITAL_entity_header
VITAL_entity_declarative_part
end [entity] [*entity*_simple_name] ;

VITAL_Level_1_architecture_body ::= [p. 35]
architecture identifier **of** *entity*_name **is**
VITAL_Level_1_architecture_declarative_part
begin
VITAL_Level_1_architecture_statement_part
end [architecture] [*architecture*_simple_name] ;

VITAL_Level_1_architecture_declarative_part ::= [p. 36]
VITAL_Level1_attribute_specification
{ VITAL_Level_1_block_declarative_item }

VITAL_Level_1_architecture_statement_part ::= [p. 36]
VITAL_Level_1_concurrent_statement { VITAL_Level_1_concurrent_statement }

VITAL_Level_1_block_declarative_item ::= [p. 36]
constant_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| VITAL_internal_signal_declaration

VITAL_Level_1_ concurrent_statement ::= [p. 36]
VITAL_wire_delay_block_statement
| VITAL_negative_constraint_block_statement
| VITAL_process_statement
| VITAL_primitive_concurrent_procedure_call

VITAL_Level1_Memory_architecture_body ::= [p. 76]
architecture identifier **of** *entity*_name **is**
VITAL_Level1_Memory_architecture_declarative_part
begin
VITAL_Level1_Memory_architecture_statement_part
end [architecture] [*architecture*_simple_name] ;

VITAL_Level1_Memory_architecture_declarative_part ::= [p. 77]
VITAL_Level1_Memory_attribute_specification
{ VITAL_Level1_Memory_block_declarative_item }

Copyright © 2001 IEEE. All rights reserved.

VITAL_Level1_Memory_architecture_statement_part ::= [p. 77]
VITAL_Level1_Memory_concurrent_statement {
VITAL_Level1_Memory_concurrent_statement }

VITAL_Level1_Memory_block_declarative_item ::= [p. 77]
constant_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| VITAL_memory_internal_signal_declaration

VITAL_Level1_Memory_concurrent_statement ::= [p. 77]
VITAL_wire_delay_block_statement
| VITAL_negative_constraint_block_statement
| VITAL_memory_process_statement
| VITAL_memory_output_drive_block_statement

VITAL_Level0_attribute_specification ::= attribute_specification [p. 7]

VITAL_Level1_attribute_specification ::= attribute_specification

VITAL_Level1_Memory_attribute_specification ::= attribute_specification [p. 76]

VITAL_library_unit ::= [p. 8]
VITAL_Level_0_entity_declaration
| VITAL_Level_0_architecture_body
| VITAL_Level_1_architecture_body
| VITAL_Level_1_memory_architecture_body

VITAL_memory_functionality_section ::= [p. 82]
{ VITAL_variable_assignment_statement | VITAL_memory_procedure_call_statement }

VITAL_memory_internal_signal_declaration ::= [p. 77]
signal identifier_list : type_mark [index_constraint] [:= expression] ;

VITAL_memory_process_declarative_item ::= [p. 79]
constant_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| VITAL_variable_declaration
| VITAL_memory_variable_declaration

VITAL_memory_process_declarative_part ::= [p. 79]
{ VITAL_memory_process_declarative_item }

VITAL_memory_process_statement ::= [p. 78]
*process*_label :
process (sensitivity_list)
VITAL_memory_process_declarative_part
begin
VITAL_memory_process_statement_part
end process [*process*_label] ;

 Copyright © 2001 IEEE. All rights reserved.

VITAL_memory_process_statement_part ::= [p. 78]
  [VITAL_memory_timing_check_section]
  [VITAL_memory_functionality_section]
  [VITAL_memory_path_delay_section]

VITAL_memory_timing_check_condition ::= *generic*_simple_name [p. 81]

VITAL_memory_timing_check_section ::= [p. 81]
  if VITAL_memory_timing_check_condition **then**
    { VITAL_memory_timing_check_statement }
  end if ;

VITAL_memory_timing_check_statement ::= procedure_call_statement [p. 81]

VITAL_memory_variable_declaration ::= [p. 79]
  variable identifier_list : type_mark [index_constraint] [:= expression] ;

VITAL_negative_constraint_block_statement ::= [p. 39]
  *block*_label :
  block
  begin
    *VITAL*_negative_constraint_block_statement_part
  end block [*block*_label] ;

VITAL_negative_constraint_block_statement_part ::= [p. 39]
  { VITAL_negative_constraint_concurrent_procedure_call
  | VITAL_negative_constraint_generate_statement _part}

VITAL_negative_constraint_concurrent_procedure_call ::= concurrent_procedure_call [p. 39]

VITAL_negative_constraint_generate_statement _part ::= [p. 39]
  VITAL_negative_constraint_generate_statement {
VITAL_negative_constraint_generate_statement}

VITAL_negative_constraint_generate_parameter_specification ::= [p. 39]
  identifier **in** *range*_attribute_name

VITAL_negative_constraint_generate_statement ::= [p. 39]
  *generate*_label :
  for VITAL_negative_constraint_generate_parameter_specification **generate**
    { VITAL_negative_constraint_concurrent_procedure_call }
  end generate {*generate*_label }

VITAL_output_drive_block_statement ::= [p. 85]
  *block*_label :
  block
  begin
    VITAL_output_drive_block_statement_part
  end block [*block*_label] ;

VITAL_output_drive_block_statement_part ::= [p. 85]
  { VITAL_primitive_concurrent_procedure_call
  | concurrent_signal_assignment_statement }

Copyright © 2001 IEEE. All rights reserved.

VITAL_primitive_concurrent_procedure_call ::= [p. 44]
*VITAL_primitive*_concurrent_procedure_call

VITAL_process_declarative_item ::= [p. 41]
constant_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| VITAL_variable_declaration

VITAL_process_declarative_part ::= [p. 41]
{ VITAL_process_declarative_item }

VITAL_process_statement ::= [p. 40]
[*process*_label :]
process (sensitivity_list)
VITAL_process_declarative_part
begin
VITAL_process_statement_part
end process [*process*_label] ;

VITAL_process_statement_part ::= [p. 42]
[VITAL_timing_check_section]
[VITAL_functionality_section]
[VITAL_path_delay_section]

VITAL_target ::= *unrestricted_variable*_name | memory_unrestricted_variable_name [p. 43, 82]

VITAL_timing_check_condition ::= *generic*_simple_name [p. 42]

VITAL_timing_check_section ::= [p. 42]
if VITAL_timing_check_condition **then**
{ VITAL_timing_check_statement }
end if ;

VITAL_timing_check_statement ::= procedure_call_statement [p. 42]

VITAL_timing_generic_declaration ::= [p. 9]
[**constant**] identifier_list ::= [**in**] type_mark [index_constraint] [:= *static*_expression] ;

VITAL_variable_assignment_statement ::= [p. 43]
VITAL_target := expression ;

VITAL_variable_declaration ::= [p. 41]
variable identifier_list : type_mark [index_constraint] [:= expression] ;

VITAL_wire_delay_block_statement ::= [p. 37]
*block*_label :
block
begin
VITAL_wire_delay_block_statement_part
end block [*block*_label] ;

 Copyright © 2001 IEEE. All rights reserved.

VITAL_wire_delay_block_statement_part ::= [p. 37]
{ VITAL_wire_delay_concurrent_procedure_call
| VITAL_wire_delay_generate_statement }

VITAL_wire_delay_concurrent_procedure_call ::= concurrent_procedure_call [p. 38]

VITAL_wire_delay_generate_parameter_specification ::= [p. 38]
identifier **in** *range*_attribute_name

VITAL_wire_delay_generate_statement ::= [p. 38]
*generate*_label :
for VITAL_wire_delay_generate_parameter_specification **generate**
{ VITAL_wire_delay_concurrent_procedure_call }
end generate [*generate*_label] ;

Copyright © 2001 IEEE. All rights reserved.

Annex B

(informative)

Glossary

This glossary contains brief, informal definitions of a number of hardware-specific terms and phrases that are used in the VITAL ASIC modeling specification. The definitions in this annex are not a part of the formal definition of the VITAL ASIC modeling specification.

B.1 access time: the delay time valid data appears in the data output bus of a memory when a memory access is occurred.

B.2 address: the pins in the ASIC memory used to access a portion or a whole memory location.

B.3 ASIC cell: the building block of an Application Specific Integrated Circuit.

B.4 ASIC memory: a storage block embedded in an ASIC designed typically using array of latches in a rectangular grid which has specific row and column addresses. e.g. a 16 x 4 memory means a memory array with 16 rows and there are 4 bits per row.

B.5 cross port access: It is a multi-port memory access (read or write) in which an address port associates itself to the data output port of another address port.

B.6 device delay: the intrinsic delay of a cell; it represents the delay associated from each input path to the given output of the cell.

B.7 hold time: the time period following a clock edge during which an input signal value may not change value.

B.8 interconnect path delay: delays on the wires which connect various instantiations of ASIC cells in a design.

B.9 multiport memories: these are memories with multiple address and data output ports. A single port memory can only perform either read or write at a given time. A dual port memory (2 address ports, one read one write) can perform read operation at one port and write operation at the other port simultaneously.

B.10 no change time: a stable interval associated with a setup or hold constraint. A signal checked against a control signal must remain stable during the setup period established before the start of the control pulse, the entire width of the pulse, and the hold period established after the pulse. Each of these stable intervals is a no change time.

B.11 output Retain time: the time period in which the data output signal of ASIC memories retains the previous value before changing to a new value after the propagation delay. Note that in the time period between the output retain time and propagation delay the output will go to an intermediate unknown state. Output retain time is also known as data hold time.

B.12 period: the time delay from the specified edge of a clock pulse to the corresponding edge of the following clock pulse.

 Copyright © 2001 IEEE. All rights reserved.

B.13 propagation delay: the time delay from the arrival of an input signal value to the appearance of a corresponding output signal value.

B.14 pulse width: the time duration for which the value of signal remains unchanged at a low or high state.

B.15 recovery time: the minimal time interval by which a change to an unasserted value on an asynchronous (set, reset) input signal must precede the clock edge.

B.16 removal time: the minimal time interval for which an asserted condition must be present on an asynchronous (set, reset) input signal, following the clock edge.

B.17 same port access: It is a multi-port memory access (read or write) in which an address port associates itself to its corresponding data output port. ASIC memories are typically designed as addressable latches. In such cases the read only and read/write ports are always associated with a corresponding data output ports. However the write only ports do not have any direct association with data output ports

B.18 setup time: the time period prior to a clock edge during which an input signal value may not change value.

B.19 skew time: the maximum allowable delay between two signals. A delay which exceeds the skew time causes devices to behave unreliably.

B.20 subword memories: a set of contiguous bits of a memory word can be accessed in this kind of memory by an associated enable pin. e.g. A 4-bit wide memory can have 2 bits per soberer - bits 3 to 2 is one soberer controlled by WEB(1) and bits 1 to 0 is another soberer controlled by WEB(0).

B.21 synchronous memories: a clock signal is needed to trigger all read and write operations in this type of memory.

Copyright © 2001 IEEE. All rights reserved.

Annex C

(informative)

Bibliography

[C1] EIA-567-A VHDL Hardware Component Modeling and Interface Standard.[3]

[C2] IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual.[4,5]

[C3] IEEE Std 1076/INT-1991, IEEE Standards Interpretations: IEEE Standard VHDL Language Reference Manual.[4]

[3]EIA publications are available from the Electronic Industries Association, 2500 Wilson Boulevard, Arlington, VA 22201-3834, USA.

[4]IEEE publications are available from the Institute of of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[5]ANSI publications are available from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

 Copyright © 2001 IEEE. All rights reserved.