WELCOME TO THE

A Learning Tool for IEEE Std. 1076, VHDL

This CD-ROM contains the IEEE VHDL Interactive Tutorial and the Microsoft® Windows® , SUN® OS and SUN Solaris® versions of the Spyglass® Mosaic(TM) browser.

To access the tutorial with your current browser click this icon:

VHDL

HOME PAGE

For information on installing Spyglass Mosaic or for more information on the viewing the tutorial click here for the README.TXT file.

IEEE

Networking the World

A Learning Tool for IEEE Std. 1076, VHDL

Software License Agreement

Copyright Information and Disclaimers

About the CD-ROM

Getting Started

Tutorial

Basic VHDL	Structural VHDL	Behavioral VHDL	System Level VHDL
MODULE 1	MODULE 2	MODULE 3	MODULE 4

IEEE Standard VHDL Language Reference Manual

(IEEE Std 1076-1993)

Introduction | Table of Contents | 

IEEE

Networking the World.

Published by the Institute of Electrical and Electronics Engineers, Inc.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. (IEEE)

SOFTWARE LICENSE AGREEMENT

IMPORTANT - READ CAREFULLY - THIS AGREEMENT DEFINES YOUR RIGHTS TO USE THIS ENCLOSED SOFTWARE PRODUCT. OPENING THIS SEALED PACKAGE OR USING THIS SOFTWARE CONSTITUTES YOUR ACCEPTANCE OF THE TERMS OF THIS AGREEMENT AND ALL RIGHTS THEREIN.

This software is owned by the Institute of Electrical and Electronics Engineers, Inc., ("Licensor") and licensed to you as Licensee ("Licensee"). If you do not agree to the following terms and conditions, return this product to Licensor and all amounts paid by you (if any) will be refunded. Failure to return this product to Licensor within thirty (30) days of its delivery to you will be deemed to constitute acceptance.

License

Licensor grants to you a non-transferable, non-exclusive license to copy into and use on a single personal computer the accompanying data file(s) in electronic format, hereinafter referred to as the "Software." Neither the Software nor any accompanying documentation may be copied in whole or in part, except for a backup or archival copy that bears all copyright notices and any other identifying or restrictive legends appearing on the Software as received. Except as expressly permitted herein, you may not copy, distribute, modify or make derivative works based on the Software nor may this Software or any backup copy or archival copy of this Software be placed on an electronic network without prior written consent of Licensor. Requests for reproduction, distribution or use in a multiple-user environment should be directed to the IEEE Standards Department, Copyrights and Permissions, 445 Hoes Lane, P.O.

Box 1331, Piscataway, New Jersey 08855-1331, USA.

Limited Warranty

LICENSOR SHALL REPLACE ANY DEFECTIVE MEDIA CONTAINING THE SOFTWARE IF SUCH DEFECTIVE MEDIA IS RETURNED TO LICENSOR WITHIN THIRTY (30) DAYS AFTER DELIVERY. THE FOREGOING WARRANTY IS THE SOLE AND EXCLUSIVE WARRANTY AND IS GIVEN IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, THE SOFTWARE IS BEING LICENSED TO YOU "AS IS," AND LICENSOR DOES NOT WARRANT THAT THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR-FREE. THE SOFTWARE IS BASED ON THE VHDL INTERACTIVE TUTORIAL, COPYRIGHT © 1996 BY THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. AND IEEE STD 1076-1993 IEEE STANDARD VHDL LANGUAGE REFERENCE MANUAL, COPYRIGHT © 1994 BY THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. IN THE EVENT OF A DISCREPANCY BETWEEN THE PUBLISHED IEEE STD 1076-1993 AND THE PORTIONS OF IT CONTAINED IN THIS SOFTWARE, THE PUBLISHED HARD COPY IS THE REFEREE DOCUMENT.

Liability

In no event shall Licensor, its employees, agents, suppliers, or contractors be liable for any damages of any kind or character, including without limitation any compensatory, incidental, direct, indirect, special, punitive or consequential damages, loss of use, loss of data, loss of income or profit, loss of or damage to property, claims of third parties, or other losses of any kind or character or attorneys' fees in connection with a claim relating to this Agreement or the performance of the Software. In the event that liability is nevertheless imposed on Licensor, its employees, agents, suppliers or contractors, the liability shall not exceed the fee paid for this Software.

In no event shall Licensor have any liability whatsoever with respect to a backup copy of the Software made by you as permitted in this Agreement.

Termination

This License shall automatically terminate in the event of a breach of the terms of this Agreement.Upon termination, you will be required to return to Licensor the software, the backup or archive copy and any accompanying documentation, and remove any copy residing

on your hard drive.

General Terms

This Agreement is not assignable or transferable. The rights under this Agreement or any License granted hereunder may not be assigned, sublicensed or otherwise transferred by Licensee without the prior written consent of Licensor. This is the entire agreement between the parties relating to the subject matter hereof and may only be modified in writing signed by each party. This Agreement supersedes any proposal or prior agreement, oral or written, and any other communications between the parties relating to the subject matter of this Agreement. This Agreement is governed under the laws of the State of New York. Any questions or comments regarding this Agreement or the Software should be directed to: Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, P.O. Box 1331, Piscataway, New Jersey 08855-1331, Attention: Standards Press.

Copyright Information and Disclaimers

Copyright Notice:

Disclaimer:

IEEE MAKES NO WARRANTIES WITH RESPECT TO THE CONTENTS OF THIS PRODUCT. THE CONTENTS ARE PROVIDED FOR USE "AS IS". IEEE DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Limitation of Liability:

In no event shall IEEE, its employees, agents, suppliers, or contractors be liable for any damages of any kind or character, including without limitation any compensatory, incidental, direct, indirect, special, punitive, or consequential damages, loss of use, loss of data, loss of income or profit, loss of or damage to property, claims of third parties, or other losses of any kind or character. In the event that liability is nevertheless imposed on IEEE, its employees, agents, suppliers, or contractors, the liability shall not exceed the one-time charge paid for this product.

License/User Information :

This product is licensed for single user purposes. Requests for multiple-user environments will be available upon request. For information, please email such requests to stds.vhdlinfo@ieee.org or call (908) 562-3804.

IEEE Standards Press:

IEEE Standards Press publications are not consensus documents. Information contained in this and other works has been obtained from sources believed to be reliable, and reviewed by credible members of IEEE Technical Societies and/or Standards Coordinating Committees. Neither the IEEE nor its authors/developers guarantee the accuracy or completeness of any information published herein, and neither the IEEE nor its authors/developers shall be responsible for any errors, omissions, or damages arising out of use of this document.

Likewise, while the author/developer and publisher believe that the information and guidance given in this work serves as an enhancement to users, all parties must rely upon their own skill and judgment when making use of it. Neither the author nor the publisher assumes any liability to anyone for any loss or damage caused by any error or omission in the work, whether such error or omission is the result of negligence or any other cause. Any and all such liability is disclaimed.

This work is published with the understanding that the IEEE and its authors/developers are supplying information through this publication, not attempting to render engineering or other

professional services. If such services are required, the assistance of an appropriate professional should be sought. The IEEE Standards Press is not responsible for any statements and/or opinions advanced in this publication.

Review Policy:

The information contained in the IEEE Standards Press publications is reviewed and evaluated by peer reviewers of relevant IEEE Technical Societies and/or Standards Coordinating Committees. The authors/developers addressed all of the reviewers' comments to the satisfaction of both the IEEE Standards Press and those who served as peer reviewers for this document.

The quality of the presentation of information contained in this publication reflects not only the obvious efforts of the authors/developers, but also the peer reviewers. The IEEE Standards Press acknowledges with appreciation their dedication and contribution of time and effort on behalf of the IEEE.

About the CD-ROM

Acknowledgements - These organizations have made this tutorial possible.

DARPA Electronics Technology Office (DARPA/ETO)
United States Air Force Wright Aeronautical Laboratory
Rapid Prototyping of Application Specific Signal Processors (RASSP)
RASSP Education and Facilitation

The Modular Tutorial portion of this CD-ROM was supported in part by the Defense Advanced Research Projects Agency Electronics Technology Office (DARPA/ETO) and the U.S. Air Force RASSP Education and Facilitation (RASSP E&F) program under contract number F33615-94-C-1457.

The following individuals are recognized with gratitude for their contributions:

Development Coordinators:

SCRA (South Carolina Research Authority)
Anthony Gadient

Jack Stinson
Tommy Taylor
Christopher Florio

Contributing Developers:

University of Virginia
James Aylor
Robert Klenke
Maximo Salinas
Ron Waxman
Georgia Institute of Technology
Vijay Madisetti
Tom Egolf
Shahram Famorzadeh
Raytheon
Dave Wilsey
Mitchell Heller
Mississippi State University
Scott Calhoun
Cassie S. Brook
Michael Mott
LaRosa J. Harris
University of Cincinnati
Harold Carter

Peer Reviewers

Dr. Paul Y.S. Cheung
Dean of Engineering
The University of Hong Kong
IEEE Asia Pacific Region Director

Dr. Vijay K. Madisetti
Associate Professor,
Electrical & Computer Engineering
Georgia Institute of Technology
President, VP Technologies, Inc.
IEEE Signal Processing Society
Liaison to IEEE Press

Dr. John Willis

FTL Systems, Inc.
IEEE Design Automation Standards Committee (DASC) Secretary
Chair, IEEE DASC Parallel Simulation Group

Special Thanks are extended to these individuals by the IEEE Standards Press for their contributions to this publication and their continued efforts towards the further development of IEEE Standards and standards-related products and activities.

Grateful acknowledgement is also extended to Anthony Gadient, Jack Stinson and Tommy Taylor for their dedicated contributions that made this IEEE Standards Press project possible.

CD-ROM Format/Browser Options

The information available on this CD-ROM has been composed in a manner thought to be compatible with most computer architectures. The final appearance of these documents, however, is dependent on the client software as implemented at the user site and as such is outside of RASSP E&F's control. This CD-ROM has been checked using HTML 2.0 browsers. For best results, use a Super-VGA or better display. For small screen displays, some pages many not fit on the screen so scrolling will be required. You may wish to maximize the size of your browser display area by adjusting some of your browser options (i.e. remove menu bars, remove URL displays, etc.). Readability may be improved by adjusting the default font size or text colors in your browser. A README.TXT file is available on the CD-ROM to help with any problems starting up the browser and accessing the home page. This CD-ROM was designed as a stand alone tutorial. There are a few links (RASSP Web Server, DARPA/ETO, etc.) that use the Internet. If you do not have an Internet connection, an error message will appear when these links are selected. However, the usefulness of the tutorial is not diminished by not having these links. Feedback on the CD-ROM may be sent to stds.vhdlinfo@ieee.org.

.

http://roza.gmu.edu/IEEE_Tutorial/: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú**¡£

.

http://roza.gmu.edu/IEEE_Tutorial/README.TXT: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔÔO¶¨µÄ¹ ‡ú¡£**

.

http://stdsbbs.ieee.org/: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú**¡£

.

http://eto.sysplan.com/ETO/RASSP/: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú**¡£

.

http://www.wl.wpafb.af.mil/welcome.html: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔÔO¶¨µÄ¹ ‡ú¡£**

.

http://rassp.scra.org/: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú**¡£

.

http://rassp.scra.org/offerings/offerings.html: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°

ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú¡£**

.

http://www.ansi.org/: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú¡£**

.

http://www.ieee.org/: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú¡£**

.

http://www.iso.ch/: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾**WÖ·ÖÐÔO¶¨µÄ¹ ‡ú**¡£

.

http://www.erc.msstate.edu/mpl/vhdl-class/html: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾WÖ·ÖÐÔO¶¨µÄ¹ ‡ú¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD4/SEC3/SSEC6/HTML/SLIDE28.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° Ëü²»ÊÇŸo·¨×xÈ¡¡¢¾ÍÊÇ”XÈ¡±»ÖÐ”àÁË¡¢»òÊÇŒ£°¸ß^ÔçÍ£Ö¹¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD4/SEC3/SSEC6/HTML/SLIDE31.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°Ëü²»ÊÇŸo·¨×xÈ¡¡¢¾ÍÊÇ”XÈ¡±»ÖÐ”àÁË¡¢»òÊÇŒ£°¸ß^ÔçÍ£Ö¹¡£

.

http://rassp.scra.org/public/atl/taxonomy.html: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
ËüµÄ¾WÓò»òÂ·•½³¬ß^†¢Ê¼¾WÖ·ÖÐÔÔO¶¨µÄ¹ ‡ú¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD1/SEC4/SSEC1/IMAGES/IMAGE12.GIF: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
´ËÒ»™n°¸Î»ì¶**Íâ²¿¾WÓò**£¬¶øÇÒëxé_ËüµÄélµÀÎ»Ö·Ì«ßh¡£Èç¹ûÄúÔö¼Ó†¢Ê¼¾WÖ·µÄÍâ²¿¾WÓò”XÈ¡Éî¶È£¬Äú¾Í¿ÉÒÔ”XÈ¡µ½´ËÒ»™n°¸¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD1/SEC4/SSEC3/IMAGES/IMAGE21A.GIF: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ´ËÒ»™n°¸Î»ì¶**Íâ²¿¾WÓò**£¬¶øÇÒëxé_ËüµÄélµÀÎ»Ö·Ì«ßh¡£Èç¹ûÄúÔö¼Ó†¢Ê¼¾WÖ·µÄÍâ²¿¾WÓò”XÈ¡Éî¶È£¬Äú¾Í¿ÉÒÔ”XÈ¡µ½´ËÒ»™n°¸¡£

.

http://rassp.scra.org/public/tb/honeywell/HONEYWELL-DOCS.html: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËüµÄ¾WÓòÊÇÍâ²¿æœ½YµÄÍâ²¿æœ½Y¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD4/SEC3/SSEC6/HTML/SLIDE29.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° Ëü²»ÊÇŸo·¨×xÈ¡¡¢¾ÍÊÇ”XÈ¡±»ÖÐ”àÁË¡¢»òÊÇŒ£°¸ß^ÔçÍ£Ö¹¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD4/SEC3/SSEC6/HTML/SLIDE30.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° Ëü²»ÊÇŸo·¨×xÈ¡¡¢¾ÍÊÇ”XÈ¡±»ÖÐ”àÁË¡¢»òÊÇŒ£°¸ß^ÔçÍ£Ö¹¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD3/SEC2/SSEC1/IMAGES/IMAGE9.GIF: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ´ËÒ»™n°¸Î»ì¶**Íâ²¿¾WÓò**£¬¶øÇÒëxé_ËüµÄélµÀÎ»Ö·Ì«ßh¡£Èç¹ûÄúÔö¼Ó†¢Ê¼¾WÖ·µÄÍâ²¿¾WÓò”XÈ¡Éî¶È£¬Äú¾Í¿ÉÒÔ”XÈ¡µ½´ËÒ»™n°¸¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD3/SEC2/SSEC2/IMAGES/IMAGE20B.GIF: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ´ËÒ»™n°¸Î»ì¶**Íâ²¿¾WÓò**£¬¶øÇÒëxé_ËüµÄélµÀÎ»Ö·Ì«ßh¡£Èç¹ûÄúÔö¼Ó†¢Ê¼¾WÖ·µÄÍâ²¿¾WÓò”XÈ¡Éî¶È£¬Äú¾Í¿ÉÒÔ”XÈ¡µ½´ËÒ»™n°¸¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD3/SEC2/SSEC2/IMAGES/IMAGE21.GIF: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
´ËÒ»™n°¸Î»ì¶**Íâ²¿¾WÓò**£¬¶øÇÒëxé_ËüµÄélµÀÎ»Ö·Ì«ßh¡£Èç¹ûÄúÔö¼Ó†¢Ê¼¾WÖ·µÄÍâ²¿¾WÓò”XÈ¡Êî¶È£¬Äú¾Í¿ÉÒÔ”XÈ¡µ½´ËÒ»™n°¸¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD3/SEC2/SSEC3/IMAGES/IMAGE26.GIF: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
´ËÒ»™n°¸Î»ì¶**Íâ²¿¾WÓò**£¬¶øÇÒëxé_ËüµÄélµÀÎ»Ö·Ì«ßh¡£Èç¹ûÄúÔö¼Ó†¢Ê¼¾WÖ·µÄÍâ²¿¾WÓò”XÈ¡Éî¶È£¬Äú¾Í¿ÉÒÔ”XÈ¡µ½´ËÒ»™n°¸¡£

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/HTML/HOME.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
ËÅ·þÆ÷^ó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/HTML/SLIDE1A.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
ËÅ·þÆ÷^ó¸æ**ÕÒÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/HTML/SLIDE1B.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬°
ËÅ·þÆ÷^ó¸æ**ÕÒÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC3/HTML/SLIDE11.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC3/HTML/SLIDE12.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC3/SSEC2/HTML/SLIDE14.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC3/SSEC2/HTML/SLIDE15.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC3/SSEC1/HTML/SLIDE13.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC1/HTML/SLIDE1.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷^ó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC1/HTML/SLIDE3.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷^ó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC3/SSEC3/HTML/SLIDE16.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷^ó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC1/HTML/SLIDE2.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷^ó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC1/HTML/SLIDE4.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷^ó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC1/HTML/SLIDE5.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC2/HTML/SLIDE6.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC2/HTML/SLIDE7.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC2/HTML/SLIDE8.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

.

http://roza.gmu.edu/IEEE_Tutorial/TUTORIAL/MOD2/NOTES/SEC2/SSEC2/HTML/SLIDE9.HTM: ¦¹¤@ÀÉ®×¨Ã¥¼³Q Teleport Pro ©ÒÂ^¨ú¡A¦]¬° ËÅ·þÆ÷ˆó¸æ**ÕÒ²»µ½´ËÒ»™n°¸¡£**

Getting Started

Tutorial Purpose

Effective top-down design using VHDL is critical in order to realize reductions in the development time and cost of complex digital electronic systems. The purpose of this interactive VHDL tutorial is to help designers learn how to effectively use VHDL to design complex digital electronic systems. To support this learning process, the tutorial is organized around four related modules that are designed to incrementally add to your understanding of the VHDL language and its application. By integrating these modules with the VHDL Language Reference Manual, IEEE Standard 1076-1993, (VHDL LRM) in a hypertext environment, we believe that this interactive tutorial will not only help you to learn the language but will provide a useful reference to you as you progress from novice to expert.

Modular Tutorial Page Layout

Each module is organized as a set of hyper-linked slides. A **MAP** link for each module is provided that presents a graphical view of the topics discussed in the module. An **INDEX** link is provided that enables hyper-linked access to any topic area or slide within a module. At the top of most pages are the **IEEE VHDL LRM** icon and the **VHDL Interactive Tutorial Home Page** icon. The linked **IEEE VHDL LRM** icon will always return you to the VHDL LRM Table of Contents and the linked **VHDL Interactive Tutorial Home Page** icon will always return you to the tutorial home page. Together these links provide easy access to any material that may be of interest. For a graphical introduction to the modular tutorial page layout and links, click here.

Sponsorship

This VHDL Interactive Tutorial was initiated as part of the RASSP Education & Facilitation (RASSP E&F) effort with support from the Defense Advanced Research Projects Agency Electronics Technology Office (DARPA/ETO) and United States Air Force Wright Aeronautical Laboratory under contract number F33615-94-C-1457. Rapid Prototyping of Application Specific Signal Processors (RASSP) is a major DARPA/Tri-Service initiative to reinvent the process by which embedded digital signal processors are developed. The goal is a four-fold reduction in the time from concept to fielded prototype on both new designs and design upgrades, with similar improvements in life cycle cost, quality and supportability. More information on RASSP may be obtained from the RASSP web site (http://rassp.scra.org).

Module Abstracts

This tutorial is organized into four modules. If you have little or no VHDL experience, the modules should be reviewed in order to provide the background material necessary for the next section. The modules are numbered for easy reference and start at one. The abstract information below provides a synopsis of each module.

The Basic VHDL module (module 1) is an introduction to the VHSIC Hardware Description Language and its fundamental concepts. VHDL is a language specifically developed to describe digital electronic hardware and its attributes. VHDL is a flexible language and can be applied to many different design situations. This language has several key advantages, including technology independence and a standard language for communication. The module describes many of the advantages of using VHDL and a short history of the language.

The Structural VHDL module (module 2) describes the use of VHDL for describing

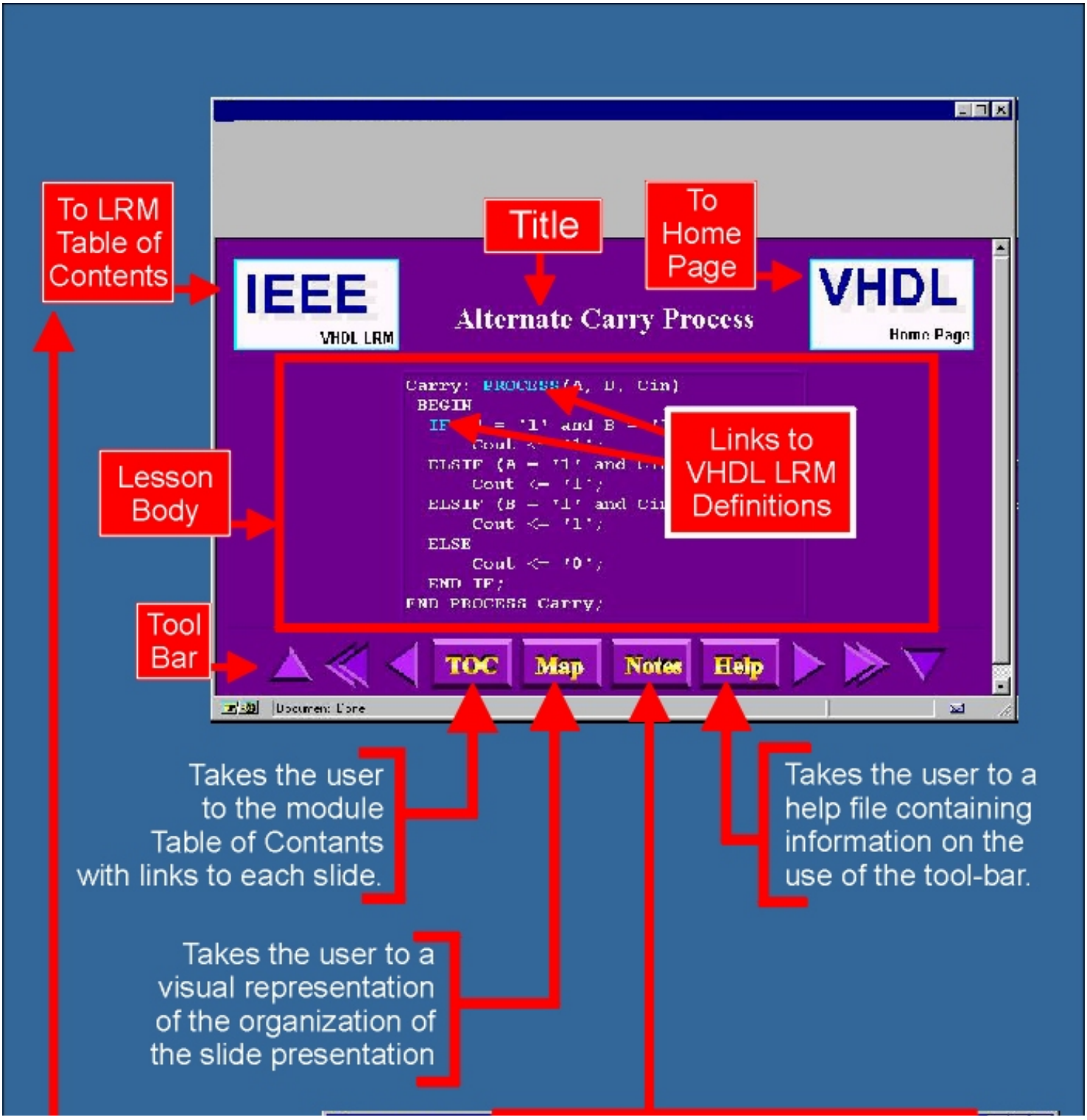
models in terms of component instantiations and interconnections. Structural VHDL can be appropriate at any level of design. For example, testbenches for completed components are often described using structural VHDL. Furthermore, structural VHDL supports the use of libraries and component reuse. This module first describes the process of creating, or instantiating, a component for simulation. A component instantiation declares a component ready for use in the architecture and specifies key parameters, if necessary. The generate statement is capable of creating regular structures automatically, such as RAM and ROM. Thus, the generate statement can eliminate some repetitiveness when dealing with such structures. Additionally, this module shows how VHDL supports libraries and component reuse. Components in structural VHDL are fully described outside the architecture, most often in component libraries. Configuration of these components involves selecting an entity and architecture for the component and specifying parameters for the component. The Structural VHDL module does not include a large, comprehensive example. However, an example highlighting the use of each VHDL construct is provided. Structural VHDL supports libraries and design partitioning through configuration; this module shows the VHDL constructs supporting these concepts.

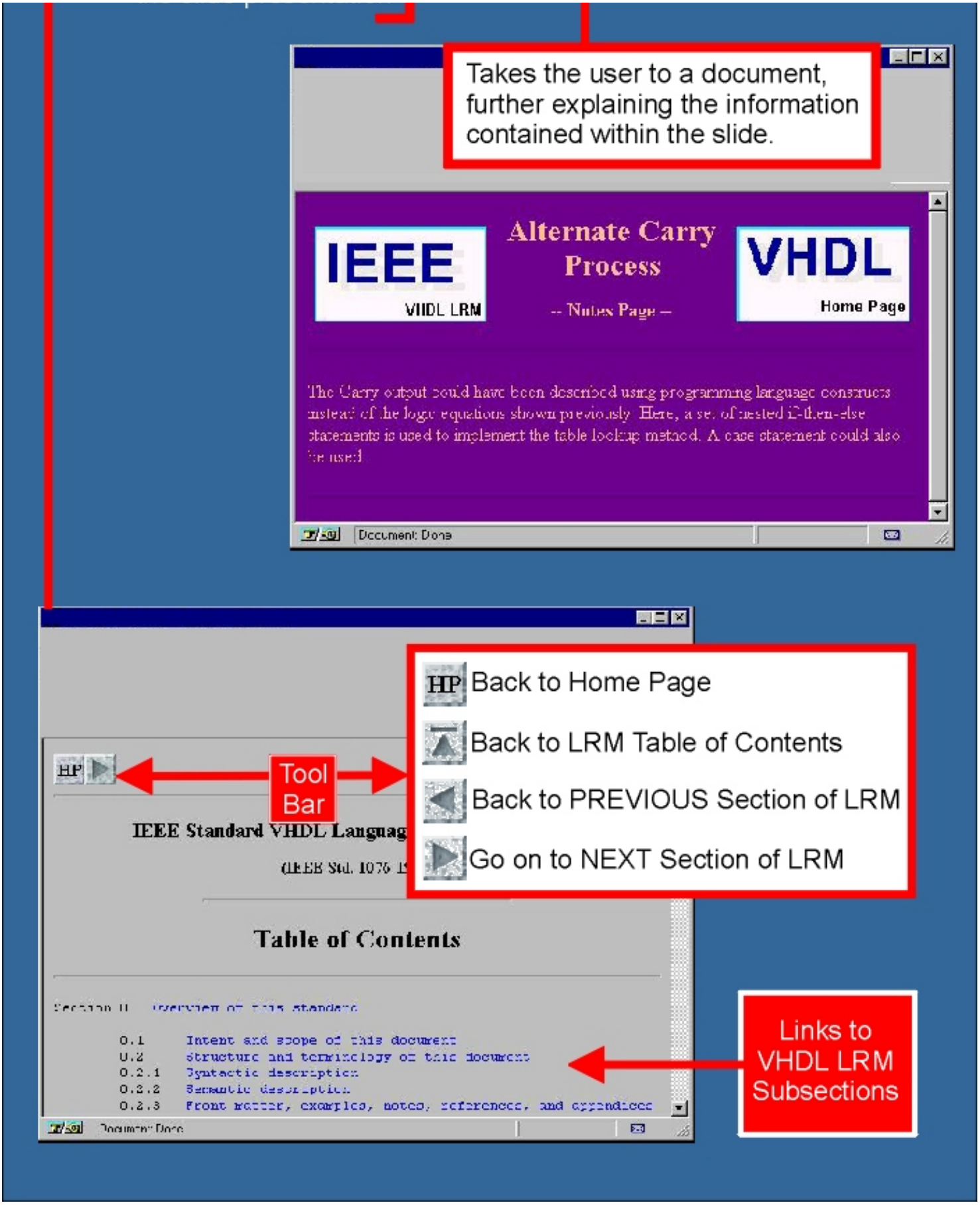
The Behavioral VHDL module (module 3) describes features of the language that describe the behavior of components in response to signals. Behavioral descriptions of hardware utilize software engineering practices and constructs to achieve a functional model. Timing information is not necessary in a behavioral description, although such information certainly can be added. The VHDL constructs in this module focus on describing hardware utilizing software engineering practices. The VHDL process construct is described first. Processes run code in a top to bottom fashion, similar to a computer program. The types of statements allowed in a process, referred to as 'sequential' statements, are listed. Subprograms are another behavioral construct, allowing for code reuse and simplification. One use of subprograms is in bus resolution functions. These important functions allow the use of buses with multiple signal drives in VHDL models. Packages are another useful VHDL feature in behavioral modeling. Packages can contain the code for subprograms as well as often used custom data types. Finally, the module describes the use of testbenches and lists some problems to avoid in VHDL. The Behavioral VHDL module ends with a comprehensive example using the SDSP microprocessor. The details of this microprocessor are not relevant to this module, but some of the underlying code is shown for instructional purposes. Several subprograms that implement basic processor functions, such as the add function, are shown. Additionally, the testbench and some control program code is shown. This example shows the many uses of behavioral VHDL.

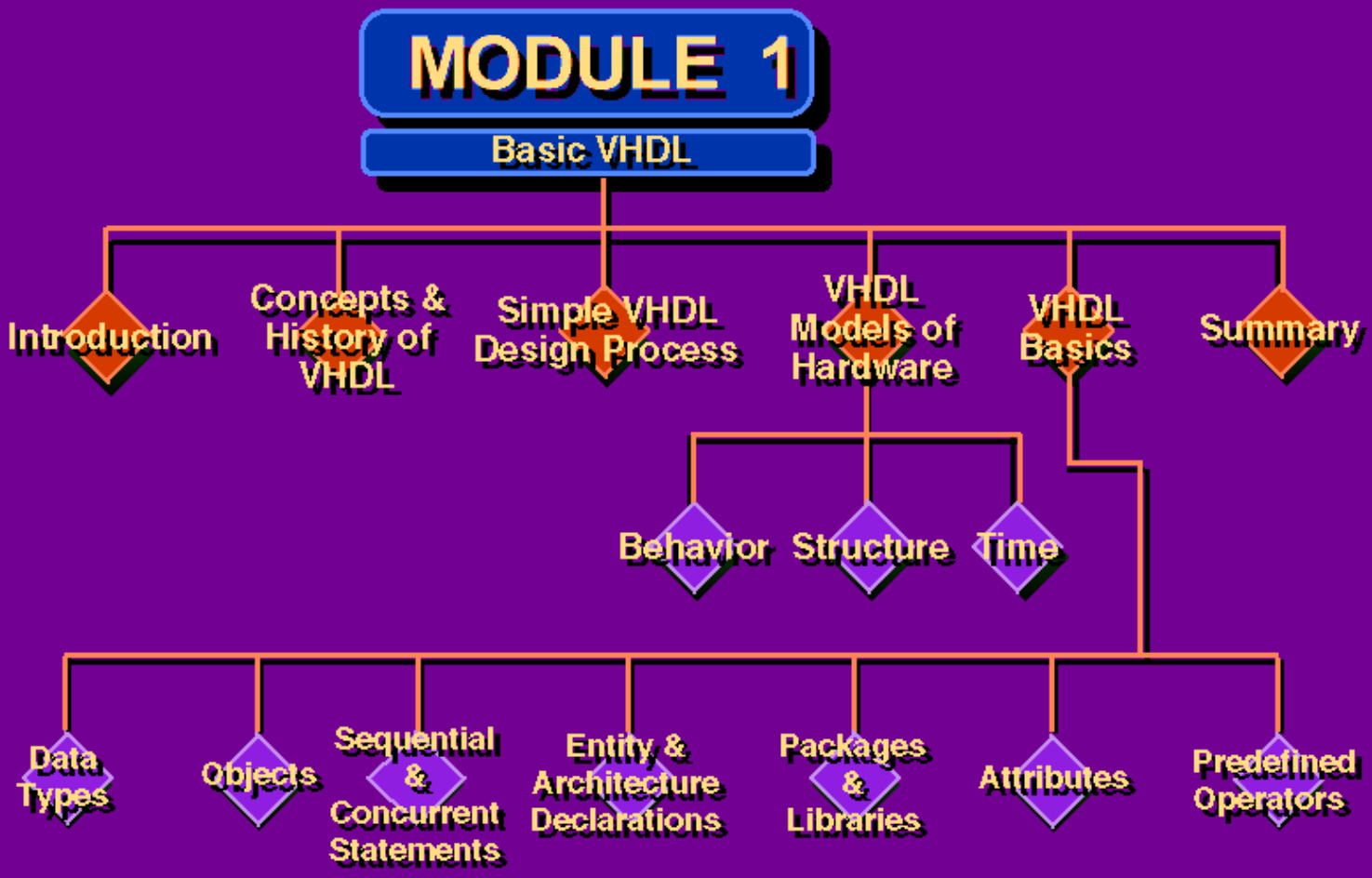
The System Level VHDL module (module 4) covers a wide range of topics, focusing on VHDL constructs as applied to higher levels of design abstraction. A definition of 'system' is presented for purposes of this module along with several key concepts. This module is not intended for the instruction of system level design; rather, this module focuses on the usefulness of VHDL at the system level. Therefore, some time is also spent on the many advantages of using VHDL at this level. This module also presents uninterpreted modeling using VHDL, as this type of modeling is frequently used at higher levels. The VHDL constructs supporting the system level design are described next. The first group of constructs provides abstract data types to the designer. Using records and aliases, the designer can implement data types, such as 'tokens', that are important for uninterpreted modeling. Shared variables allow for sharing data among processes and the TEXTIO package allows the design to process files for input and output. Finally, the use of VHDL in object oriented design is shown. Using VHDL in object oriented design has several advantages over other methods. Finally, two comprehensive examples showing the use of VHDL at the system level are provided. The first example, using the University of Virginia's ADEPT system, shows the use of records and abstract data types to implement Petri Net models using VHDL. Extensive use of functions and procedures shows the power of subprograms in VHDL. The Honeywell PML provides another example of system level VHDL. The Honeywell PML models computer components at the uninterpreted level and can consider various bus types. Together, these examples show the use of VHDL in a complex system level design.

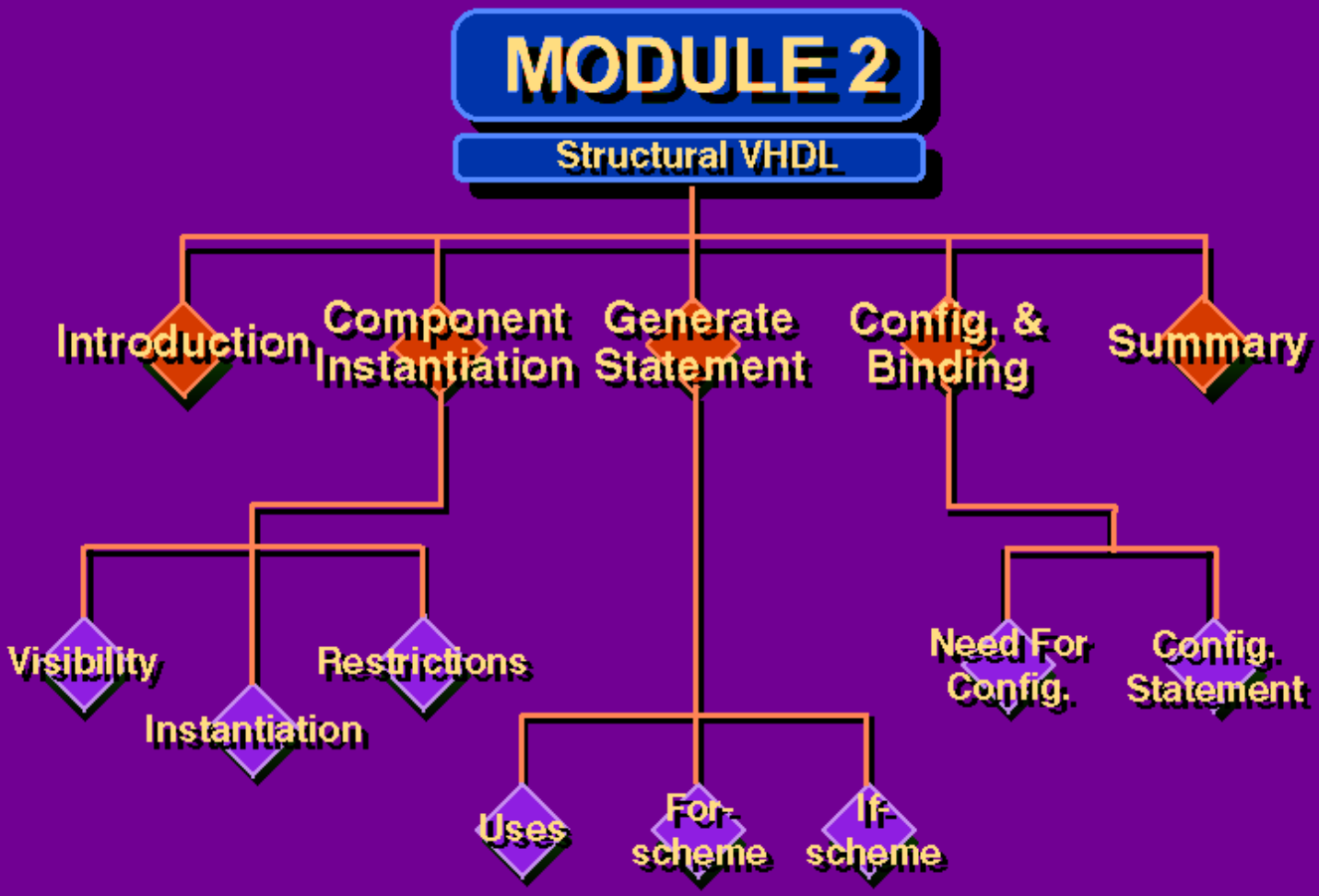
VHDL Interactive Tutorial

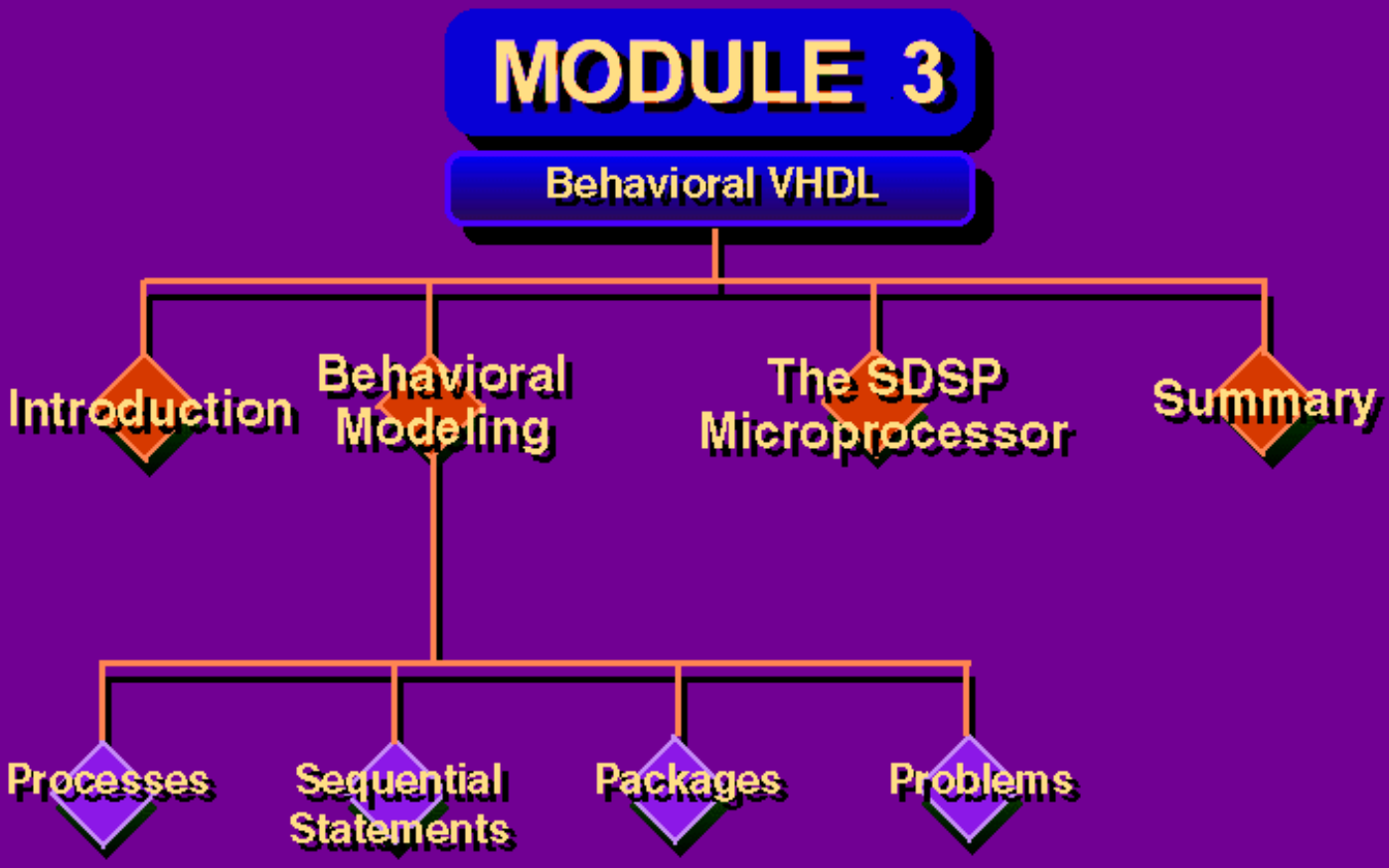
Each slide consists of three primary elements, a title, a lesson body and a tool bar. The tool bar provides access to previous and next slides, previous and next sections, to notes pages and to the map and index. A picture of a typical module slide is presented below.

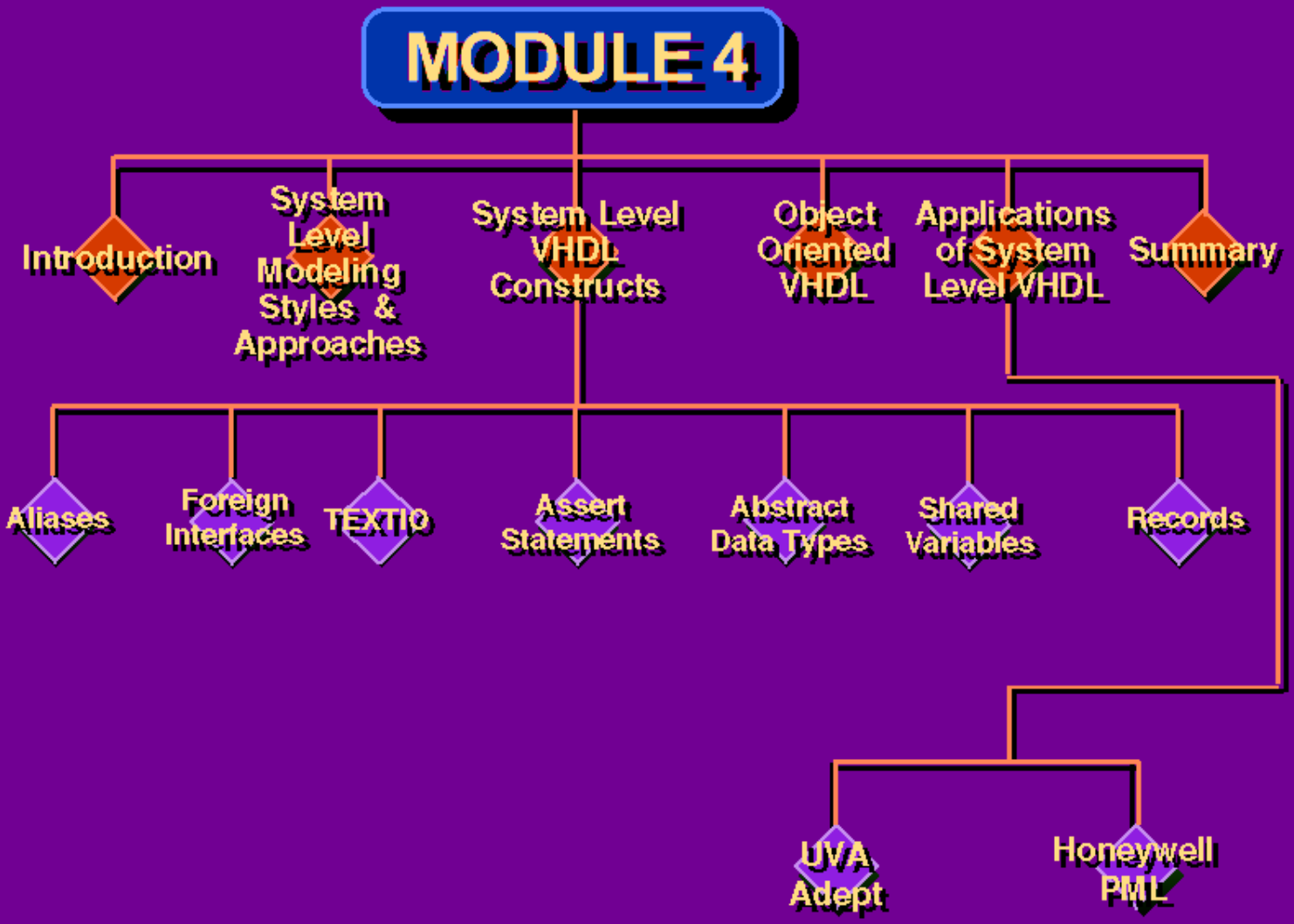



Takes the user to a document, further explaining the information contained within the slide.
IEEE
VHDL LRM
Alternate Carry Process
-- Notes Page --
VHDL
Home Page
The Carry output could have been described using programming language constructs instead of the logic equations shown previously. Here, a set of nested if-then-else statements is used to implement the table lookup method. A case statement could also be used.
Document: Done
HP Back to Home Page
Back to LRM Table of Contents
Back to PREVIOUS Section of LRM
Go on to NEXT Section of LRM
HP
Tool Bar
IEEE Standard VHDL Languag
(IEEE Std. 1076
Table of Contents
0.1 Intent and scope of this document
0.2 Structure and terminology of this document
0.2.1 Syntactic description
0.2.2 Semantic description
0.2.3 Front matter, examples, notes, references, and appendices
Links to VHDL LRM Subsections


MODULE 1
Basic VHDL
Introduction
Concepts & History of VHDL
Simple VHDL Design Process
VHDL Models of Hardware
VHDL Basics
Summary
Behavior
Structure
Time
Data Types
Objects
Sequential & Concurrent Statements
Entity & Architecture Declarations
Packages & Libraries
Attributes
Predefined Operators


MODULE 2
Structural VHDL
Introduction
Component Instantiation
Generate Statement
Config. & Binding
Summary
Visibility
Instantiation
Restrictions
Need For Config.
Config. Statement
Uses
For-scheme
If-scheme


MODULE 3
Behavioral VHDL
Introduction
Behavioral Modeling
The SDSP Microprocessor
Summary
Processes
Sequential Statements
Packages
Problems


MODULE 4
Introduction
System Level Modeling Styles & Approaches
System Level VHDL Constructs
Object Oriented VHDL
Applications of System Level VHDL
Summary
Aliases
Foreign Interfaces
TEXTIO
Assert Statements
Abstract Data Types
Shared Variables
Records
UVA Adept
Honeywell PML

Basic VHDL - Module 1

IEEE VHDL LRM

VHDL HOME PAGE

Table of Contents

IEEE Standard VHDL Language Reference Manual

(IEEE Std. 1076-1993)

Table of Contents

IEEE Standard VHDL Language Reference Manual

(IEEE Std. 1076-1993)

Index

Section 3

Types

This section describes the various categories of types that are provided by the language as well as those specific types that are predefined. The declarations of all predefined types are contained in package STANDARD, the declaration of which appears in Section 14.

A type is characterized by a set of values and a set of operations. The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type. The remaining operations of a type are the basic operations and the predefined operators (see 7.2). These operations are each implicitly declared for a given type declaration immediately after the type declaration and before the next explicit declaration, if any.

A *basic operation* is an operation that is inherent in one of the following:

-- An assignment (in assignment statements and initializations)

-- An allocator

-- A selected name, an indexed name, or a slice name

-- A qualification (in a qualified expression), an explicit type conversion, a formal or actual part in the form of a type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type

-- A numeric literal (for a universal type), the literal **null** (for an access type), a string literal, a bit string literal, an aggregate, or a predefined attribute

There are four classes of types. *Scalar* types are integer types, floating point types, physical types, and types defined by an enumeration of their values; values of these types have no elements. *Composite* types are array and record types; values of these types consist of element values. *Access* types provide access to objects of a given type. *File* types provide access to objects that contain a sequence of values of a given type.

The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case where the constraint imposes no restriction is also included); a value is said to *satisfy* a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint. A value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself; such a subtype is said to be *unconstrained* (it corresponds to a condition that imposes no restriction). The base type of a type is the type itself.

The set of operations defined for a subtype of a given type includes the operations defined for the type; however, the assignment operation to an object having a given subtype only assigns values that belong to the subtype. Additional operations, such as qualification (in a qualified expression) are implicitly defined by a subtype declaration.

The term *subelement* is used in this manual in place of the term element to indicate either an element, or an element of another element or subelement. Where other subelements are excluded, the term *element* is used instead.

A given type must not have a subelement whose type is the given type itself.

A *member* of an object is either

- A slice of the object,

- A sub element of the object, or

- A slice of a sub eleent of the object

The name of a class of types is used in this manual as a qualifier for objects and values that have a type of the class considered. For example, the term *array object* is used for an object whose type is an array type; similarly, the term *access value* is used for a value of an access type.

NOTE--The set of values of a subtype is a subset of the values of the base type. This subset need not be a proper subset.

3.1 Scalar Types

Scalar types consist of *enumeration types, integer types, physical types*, and *floating point types*. Enumeration types and integer types are called *discrete* types. Integer types, floating point types,and physical types are called *numeric* types. All scalar types are ordered; that is, all relational operators are predefined for their values. Each value of a discrete or physical type has a position number that is an integer value.

```
scalar_type_definition ::=
      enumeration_type_definition | integer_type_definition
    | floating_type_definition    |physical_type_definition

range_constraint ::=  range range

range ::=
      range_attribute_name
    | simple_expression direction simple_expression

direction ::=  to | downto
```

A range specifies a subset of values of a scalar type. A range is said to be a *null* range if the specified subset is empty.

The range L **to** R is called an *ascending* range; if L > R, then the range is a null range. The range L **downto** R is called a *descending* range; if L < R, then the range is a null range. The smaller of L and R is called the *lower bound*, and the larger, the *upper bound*, of the range. The value V is said to *belong to the range* if the relations (*lower bound* <= V) and (V <= *upper bound*) are both true and the range is not a null range. The operators>, <, and <= in the preceding definitions are the predefined operators of the applicable scalar type.

For values of discrete or physical types, a value V1 is said to be *to the left of* a value V2 within a given range if both V1 and V2 belong to the range and either the range is an ascending range and V2 is the successor of V1 or the range is a descending range and V2 is the predecessor of V1. A list of values of a given range is in *left to right order* if each value in the list is to the left of the next value in the list within that range, except for the last value in the list.

If a range constraint is used in a subtype indication, the type of the expressions (likewise, of the bounds of a range attribute) must be the same as the base type of the type mark of the subtype indication. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype or if the range constraint defines a null range. Otherwise, the range constraint is not compatible with the subtype.

The direction of a range constraint is the same as the direction of its range.

NOTE--Indexing and iteration rules use values of discrete types.

3.1.1 Enumeration types

An enumeration type definition defines an enumeration type.

```
enumeration_type_definition ::=
   ( enumeration_literal { , enumeration_literal } )

enumeration_literal ::=  identifier | character_literal
```

The identifiers and character literals listed by an enumeration type definition must be distinct within the enumeration type definition. Each enumeration literal is the declaration of the corresponding enumeration literal; for the purpose of determining the parameter and result type profile of an enumeration literal, this declaration is equivalent to the declaration of a parameterless function whose designator is the same as the enumeration literal and whose result type is the same as the enumeration type.

An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character literal.

Each enumeration literal yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each additional enumeration literal is one more than that of its predecessor in the list.

If the same identifier or character literal is specified in more than one enumeration type definition, the corresponding literals are said to be *overloaded*. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal is determined according to the rules for overloaded subprograms (see 2.3).

Each enumeration type definition defines an ascending range.

Examples:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING, FALLING, AMBIGUOUS) ;

type BIT is ('0','1') ;

type SWITCH_LEVEL is ('0','1','X') ;       --  Overloads '0' and '1'
```

3.1.1.1 Predefined enumeration types

The predefined enumeration types are CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS.

The predefined type CHARACTER is a character type whose values are the 256 characters of the ISO 8859-1 character set. Each of the 191 graphic characters of this character set is denoted by the corresponding character literal.

The declarations of the predefined types CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS appear in package STANDARD in Section 14.

NOTES

1--The first 17 nongraphic elements of the predefined type CHARACTER (from NUL through DEL) are the ASCII abbreviations for the nonprinting characters in the ASCII set (except for those noted in Section 14). The ASCII names are chosen as ISO 8859-1 does not assign them abbreviations. The next 16 (C128through C159) are also not assigned abbreviations, so names unique to VHDL are assigned.

2--Type BOOLEAN can be used to model either active high or active low logic depending on the particular conversion functions chosen to and from type BIT.

3.1.2 Integer types

An integer type definition defines an integer type whose set of values includes those of the specified range.

```
integer_type_definition ::=  range_constraint
```

An integer type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the integer type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in an integer type definition must be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Integer literals are the literals of an anonymous predefined type that is called *universal_integer* in this standard. Other integer types have no literals. However, for each integer type there exists an implicit conversion that converts a value of type *universal_integer* into the corresponding value (if any) of the integer type (see 7.3.5).

The position number of an integer value is the corresponding value of the type *universal_integer*.

The same arithmetic operators are predefined for all integer types (see 7.2). It is an error if the execution of such an operation (in particular, an implicit conversion) cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type).

An implementation may restrict the bounds of the range constraint of integer types other than type *universal_integer*. However, an implementation must allow the declaration of any integer type whose range is wholly contained within the bounds -2147483647 and +2147483647 inclusive.

Examples:

```
type TWOS_COMPLEMENT_INTEGER is range -32768 to 32767;

type BYTE_LENGTH_INTEGER is range 0 to 255;

type WORD_INDEX is range 31 down to 0;

subtype HIGH_BIT_LOW is BYTE_LENGTH_INTEGER range 0 to 127;
```

3.1.2.1 Predefined integer types

The only predefined integer type is the type INTEGER. The range of INTEGER is implementation dependent, but it is guaranteed to include the range -2147483647to +2147483647. It is defined with an ascending range.

NOTE--The range of INTEGER in a particular implementation may be determined from the 'LOW and 'HIGH attributes.

3.1.3 Physical types

Values of a physical type represent measurements of some quantity. Any value of a physical type is an integral multiple of the primary unit of measurement for that type.

```
physical_type_definition ::=
   range_constraint
       units
           primary_unit_declaration
           { secondary_unit_declaration }
       end units [ physical_type_simple_name ]
primary_unit_declaration ::=  identifier

secondary_unit_declaration ::=  identifier = physical_literal ;

physical_literal ::=  [ abstract_literal ] unit_name
```

A physical type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the physical type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a physical type definition must be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Each unit declaration (either the primary unit declaration or a secondary unit declaration) defines a *unit name*. Unit names declared in secondary unit declarations must be directly or indirectly defined in terms of integral multiples of the primary unit of the type declaration in which they appear. The position numbers of unit names need not lie within the range specified by the range constraint.

If a simple name appears at the end of a physical type declaration, it must repeat the identifier of the type declaration in which the physical type definition is included.

The abstract literal portion (if present) of a physical literal appearing in a secondary unit declaration must be an integer literal.

A physical literal consisting solely of a unit name is equivalent to the integer 1 followed by the unit name.

There is a position number corresponding to each value of a physical type. The position number of the value corresponding to a unit name is the number of primary units represented by that unit name. The position number of the value corresponding to a physical literal with an abstract literal part is the largest integer that is not greater than the product of the value of the abstract literal and the position number of the accompanying unit name.

The same arithmetic operators are predefined for all physical types (see 7.2). It is an error if the execution of such an operation cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the physical type).

An implementation may restrict the bounds of the range constraint of a physical type. However, an implementation must allow the declaration of any physical type whose range is wholly contained within the bounds -2147483647 and +2147483647 inclusive.

Examples:

```
type DURATION is range -1E18 to 1E18
    units
        fs;                     -- femtosecond
        ps   =  1000 fs;        -- picosecond
        ns   =  1000 ps;        -- nanosecond
        us   =  1000 ns;        -- microsecond
        ms   =  1000 us;        -- millisecond
```

```
        sec  =  1000 ms;          -- second
        min  =  60 sec;           -- minute
    end units;

type DISTANCE is range 0 to 1E16
     units
     -- primary unit:
        A;                        -- angstrom

     -- metric lengths:
        nm   =  10 A;            -- nanometer
        um   =  1000 nm;         -- micrometer (or micron)
        mm   =  1000 um;         -- millimeter
        cm   =  10 mm;           -- centimeter
        m    =  1000 mm;         -- meter
        km   =  1000 m;          -- kilometer

     -- English lengths:
        mil  =  254000 A;        -- mil
        inch =  1000 mil;        -- inch
        ft   =  12 inch;         -- foot
        yd   =  3 ft;            -- yard
        fm   =  6 ft;            -- fathom
        mi   =  5280 ft;         -- mile
        lg   =  3 mi;            -- league
    end units DISTANCE;

variable x: distance;  variable y: duration;  variable z: integer;

x := 5 A + 13 ft - 27 inch;
y := 3 ns + 5 min;
z := ns / ps;
x := z * mi;
y := y/10;
z := 39.34 inch / m;
```

NOTES

1-- The 'POS and 'VAL attributes may be used to convert between abstract values and physical values.

2-- The value of a physical literal whose abstract literal is either the integer value zero or the floating point value zero is the same value (specifically zero primary units) no matter what unit name follows the abstract literal.

3.1.3.1 Predefined physical types

The only predefined physical type is type TIME. The range of TIME is implementation dependent, but it is guaranteed to include the range -2147483647 to +2147483647. It is defined with an ascending range. All specifications of delays and pulse rejection limits must be of type TIME. The declaration of type TIME appears in package STANDARD in Section 14.

By default, the primary unit of type TIME (1 femtosecond) is the *resolution limit* for type TIME. Any TIME value whose absolute value is smaller than this limit is truncated to zero (0) time units. An implementation may allow a given execution of a model (see 12.6) to select a secondary unit of type TIME as the resolution limit. Furthermore, an implementation may restrict the precision of the representation of values of type TIME and the results of expressions of type TIME, provided that values as small as the resolution limit are representable within those restrictions. It is an error if a given unit of type TIME appears anywhere within the design hierarchy defining a model to be executed, and if the position number of that unit is less than that of the secondary unit selected as the resolution limit for type TIME during the execution of the model.

NOTE--By selecting a secondary unit of type TIME as the resolution limit for type TIME, it may be possible to simulate for a longer period of simulated time, with reduced accuracy, or to simulate with greater accuracy for a shorter period of simulated time.

Cross-References: Delay and rejection limit in a signal assignment, 8.4 ; Disconnection, delay of a guarded signal, 5.3 ; Function NOW, 14.2 ;Predefined attributes, functions of TIME, 14.1 ; Simulation time, 12.6.2 and 12.6.3 ; Type TIME, 14.2 ; Updating a projected waveform, 8.4.1 ; Wait statements, timeout clause in, 8.1 .

3.1.4 Floating point types

Floating point types provide approximations to the real numbers. Floating point types are useful for models in which the precise characterization of a floating point calculation is not important or not determined.

```
floating_type_definition ::=  range_constraint
```

A floating type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the floating type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a floating type definition must be a locally static expression of some floating point type, but the two bounds need not have the same floating point type. (Negative bounds are allowed.)

Floating point literals are the literals of an anonymous predefined type that is called *universal_real* in this standard. Other floating point types have no literals. However, for each floating point type there exists an implicit conversion that converts a value of type *universal_real* into the corresponding value (if any) of the floating point type (see 7.3.5).

The same arithmetic operations are predefined for all floating point types (see 7.2). A design is erroneous if the execution of such an operation cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the floating point type).

An implementation may restrict the bounds of the range constraint of floating point types other than type *universal_real*. However, an implementation must allow the declaration of any floating point type whose range is wholly contained within the bounds - 1.0 E38 and + 1.0 E38 inclusive. The representation of floating point types must include a minimum of six decimal digits of precision.

NOTE--An implementation is not required to detect errors in the execution of a predefined floating point arithmetic operation, since the detection of overflow conditions resulting from such operations may not be easily accomplished on many host systems.

3.1.4.1 Predefined floating point types

The only predefined floating point type is the type REAL. The range of REAL is host-dependent, but it is guaranteed to include the range - 1.0 E38 to + 1.0 E38 inclusive. It is defined with an ascending range.

NOTE--The range of REAL in a particular implementation may be determined from the 'LOW and 'HIGH attributes.

3.2 Composite types

Composite types are used to define collections of values. These include both arrays of values (collections of values of a homogeneous type) and records of values (collections of values of potentially heterogeneous types).

```
composite_type_definition ::=
     array_type_definition
   | record_type_definition
```

An object of a composite type represents a collection of objects, one for each element of the composite object. A composite type may only contain elements that are of scalar, composite, or access types; elements of file types are not allowed in a composite type. Thus an object of a composite type ultimately represents a collection of objects of scalar or access types, one for each noncomposite subelement of the composite object.

3.2.1 Array types

An array object is a composite object consisting of elements that have the same subtype. The name for an element of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of its elements.

```
array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
    array ( index_subtype_definition { , index_subtype_definition } )
        of element_subtype_indication

constrained_array_definition ::=
    array index_constraint of element_subtype_indication

index_subtype_definition ::=  type_mark range <>

index_constraint ::=  ( discrete_range { , discrete_range } )

discrete_range ::=  discrete_subtype_indication | range
```

An array object is characterized by the number of indices (the dimensionality of the array); the type, position, and range of each index; and the type and possible constraints of the elements. The order of the indices is significant.

A one-dimensional array has a distinct element for each possible index value. A multidimensional array has a distinct element for each possible sequence of index values that can be formed by selecting one value for each index (in the given order). The possible values for a given index are all the values that belong to the corresponding range; this range of values is called the *index range*.

An unconstrained array definition defines an array type and a name denoting that type. For each object that has the array type, the number of indices, the type and position of each index, and the subtype of the elements are as in the type definition. The *index subtype* for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition. The values of the left and right bounds of each index range are not defined but must belong to the corresponding index subtype; similarly, the direction of each index range is not defined. The symbol <> (called a *box*) in an index subtype definition stands for an undefined range (different objects of the type need not have the same bounds and direction).

A constrained array definition defines both an array type and a subtype of this type:

-- The array type is an implicitly declared anonymous type; this type is defined by an (implicit) unconstrained array definition, in which the element subtype indication is that of the constrained array definition and in which the type mark of each index subtype definition denotes the subtype defined by the corresponding discrete range.

-- The array subtype is the subtype obtained by imposition of the index constraint on the array type.

If a constrained array definition is given for a type declaration, the simple name declared by this declaration denotes the array subtype.

The direction of a discrete range is the same as the direction of the range or the discrete subtype indication that defines the discrete range. If a subtype indication appears as a discrete range, the subtype indication must not contain a resolution function.

Examples:

```
--Examples of constrained array declarations:

     type MY_WORD is array (0 to 31) of BIT ;
         --  A memory word type with an ascending range.

     type DATA_IN is array (7 downto 0) of FIVE_LEVEL_LOGIC ;
         --  An input port type with a descending range.

--Example of unconstrained array declarations:

     type MEMORY is array (INTEGER range <>) of MY_WORD ;
         --  A memory array type.

--Examples of array object declarations:

     signal DATA_LINE : DATA_IN ;
         --  Defines a data input line.

     variable MY_MEMORY : MEMORY (0 to 2^n-1) ;
         --  Defines a memory of 2^n 32-bit words.
```

NOTE--The rules concerning constrained type declarations mean that a type declaration with a constrained array definition such as

```
type T is array (POSITIVE range MINIMUM to MAX) of ELEMENT;
```

is equivalent to the sequence of declarations

```
subtype index_subtype is POSITIVE range MINIMUM to MAX;
type array_type is array (index_subtype range <>) of ELEMENT;
subtype T is array_type (index_subtype);
```

where *index_subtype* and *array_type* are both anonymous. Consequently, T is the name of a subtype and all objects declared with this type mark are arrays that have the same index range.

3.2.1.1 Index constraints and discrete ranges

An index constraint determines the index range for every index of an array type and, thereby, the corresponding array bounds.

For a discrete range used in a constrained array definition and defined by a range, an implicit conversion to the predefined type INTEGER is assumed if each bound is either a numeric literal or an attribute, and if the type of both bounds (prior to the implicit conversion) is the type *universal_integer*. Otherwise, both bounds must be of the same discrete type, other than *universal_integer*; this type must be determined independently of the context, but using the fact that the type must be discrete and that both bounds must have the same type. These rules apply also to a discrete range used in an iteration scheme (see 8.9) or a generation scheme (see 9.7).

If an index constraint appears after a type mark in a subtype indication, then the type or subtype denoted by the type mark must not already impose an index constraint. The type mark must denote either an unconstrained array type or an access type whose designated type is such an array type. In either case, the index constraint must provide a discrete range for each index of the array type, and the type of each discrete range must be the same as that of the corresponding index.

An index constraint is *compatible* with the type denoted by the type mark if and only if the constraint defined by each discrete range is compatible with the corresponding index subtype. If any of the discrete ranges defines a null range, any array thus constrained is a *null array*, having no components. An array value *satisfies* an index constraint if at each index position the array value and the index constraint have the same index range. (Note, however, that assignment and certain other operations on arrays involve an implicit subtype conversion.)

The index range for each index of an array object is determined as follows:

-- For a variable or signal declared by an object declaration, the subtype indication of the corresponding object declaration must define a constrained array subtype (and thereby, the index range for each index of the object). The same requirement exists for the subtype indication of an element declaration, if the type of the record element is an array type, and for the element subtype indication of an array type definition, if the type of the array element is itself an array type.

-- For a constant declared by an object declaration, the index ranges are defined by the initial value, if the subtype of the constant is unconstrained; otherwise, they are defined by this subtype (in which case the initial value is the result of an implicit subtype conversion).

-- For an attribute whose value is specified by an attribute specification, the index ranges are defined by the expression given in the specification, if the subtype of the attribute is unconstrained; otherwise, they are defined by this subtype (in which case the value of the attribute is the result of an implicit subtype conversion).

-- For an array object designated by an access value, the index ranges are defined by the allocator that creates the array object (see 7.3.6).

-- For an interface object declared with a subtype indication that defines a constrained array subtype, the index ranges are defined by that subtype.

-- For a formal parameter of a subprogram that is of an unconstrained array type and that is associated in whole (see 4.3.2.2), the index ranges are obtained from the corresponding association element in the applicable subprogram call.

-- For a formal parameter of a subprogram that is of an unconstrained array type and whose subelements are associated individually (see 4.3.2.2), the index ranges are obtained as follows:

The directions of the index ranges of the formal parameter are that of the type of the formal; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the formal.

-- For a formal generic or a formal port of a design entity or of a block statement that is of an unconstrained array type and that is associated in whole, the index ranges are obtained from the corresponding association element in the generic map aspect (in the case of a formal generic) or port map aspect (in the case of a formal port) of the applicable (implicit or explicit) binding indication.

-- For a formal generic or a formal port of a design entity or of a block statement that is of an unconstrained array type and whose subelements are associated individually, the index ranges are obtained as follows:

The directions of the index ranges of the formal generic or formal port are that of the type of the formal; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the formal.

-- For a local generic or a local port of a component that is of an unconstrained array type and that is associated in whole, the index ranges are obtained from the corresponding association element in the generic map aspect (in the case of a local generic) or port map aspect (in the case of a local port) of the applicable component instantiation statement.

-- For a local generic or a local port of a component that is of an unconstrained array type and whose subelements are associated individually, the index ranges are obtained as follows:

The directions of the index ranges of the local generic or local port are that of the type of the local; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the local.

If the index ranges for an interface object, or member of an interface object, are obtained from the corresponding association element (when associating in whole) or elements (when associating individually), then they are determined either by the actual part(s) or by the formal part(s) of the association element(s), depending upon the mode of the interface object, as follows:

-- For an interface object or member of an interface object whose mode is **in**, **inout**, or **linkage**, if the actual part includes a conversion function or a type conversion, then the result type of that function or the type mark of the type conversion must be a constrained array subtype, and the index ranges are obtained from this constrained subtype;otherwise, the index ranges are obtained from the object or value denoted by the actual designator(s).

-- For an interface object or member of an intercace object whose mode is **out**, **buffer**, **inout**,or **linkage**, if the formal part includes a conversion function or a type conversion, then the parameter subtype of that function or the type mark of the type conversion must be a constrained array subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object denoted by the actual designator(s).

For an interface object of mode **inout** or **linkage**, the index ranges determined by the first rule must be identical to the index ranges determined by the second rule.

Examples:

```
     type Word is array (NATURAL range <>) of BIT;
     type Memory is array (NATURAL range <>) of Word (31 downto 0);

     constant A_Word: Word := "10011";
         --  The index range of A_Word is 0 to 4

     entity E is
          generic (ROM: Memory);
          port (Op1, Op2: in Word;  Result: out Word);
     end entity E;
         --  The index ranges of the generic and the ports are defined by the actuals
associated
         --  with an instance bound to E; these index ranges are accessible via the
predefined
         --  array attributes (see 14.1  ).

     signal A, B: Word (1 to 4);
     signal C: Word (5 downto 0);

     Instance: entity E
          generic map ((1 to 2) => (others => '0'))
          port map (A, Op2(3 to 4) => B (1 to 2), Op2(2) => B (3), Result => C (3
downto 1));
                --  In this instance, the index range of ROM is 1 to 2 (matching that
of the actual),
```

```
                --  The index range of Op1  is 1 to 4 (matching the index range of A),
the index range
                --  of Op2 is 2 to 4, and the index range of Result is (3 downto 1)
                --  (again matching the index range of the actual).
```

3.2.1.2 Predefined array types

The predefined array types are STRING and BIT_VECTOR, defined in package STANDARD in Section 14.

The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE:

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH ;
type STRING is array (POSITIVE range <>) of CHARACTER ;
```

The values of the predefined type BIT_VECTOR are one-dimensional arrays of the predefined type BIT, indexed by values of the predefined subtype NATURAL:

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH ;
type BIT_VECTOR is array (NATURAL range <>) of BIT ;
```

Examples:

```
variable MESSAGE : STRING(1 to 17) := "THIS IS A MESSAGE" ;

signal LOW_BYTE : BIT_VECTOR (0 to 7) ;
```

3.2.2 Record types

A record type is a composite type, objects of which consist of named elements. The value of a record object is a composite value consisting of the values of its elements.

```
record_type_definition ::=
    record
         element_declaration
        { element_declaration }
    end record [ record_type_simple_name ]

element_declaration ::=
    identifier_list : element_subtype_definition ;

    identifier_list ::=  identifier { , identifier }

    element_subtype_definition ::=  subtype_indication
```

Each element declaration declares an element of the record type. The identifiers of all elements of a record type must be distinct. The use of a name that denotes a record element is not allowed within the record type definition that declares the element.

An element declaration with several identifiers is equivalent to a sequence of single element declarations. Each single element declaration declares a record element whose subtype is specified by the element subtype definition.

If a simple name appears at the end of a record type declaration, it must repeat the identifier of the type declaration in which the record type definition is included.

A record type definition creates a record type; it consists of the element declarations in the order in which they appear in the type definition.

Example:

```
type DATE is
    record
        DAY    :INTEGER range 1 to 31;
        MONTH  :MONTH_NAME;
        YEAR   :INTEGER range 0 to 4000;
    end record;
```

3.3 Access types

An object declared by an object declaration is created by the elaboration of the object declaration and is denoted by a simple name or by some other form of name. In contrast, objects that are created by the evaluation of allocators (see 7.3.6) have no simple name. Access to such an object is achieved by an *access value* returned by an allocator; the access value is said to *designate* the object.

```
access_type_definition ::=  access subtype_indication
```

For each access type, there is a literal **null** that has a null access value designating no object at all. The null value of an access type is the default initial value of the type. Other values of an access type are obtained by evaluation of a special operation of the type, called an *allocator*. Each such access value designates an object of the subtype defined by the subtype indication of the access type definition. This subtype is called the *designated subtype* and the base type of this subtype is called the *designated type*. The designated type must not be a file type.

An object declared to be of an access type must be an object of class variable. An object designated by an access value is always an object of class variable.

The only form of constraint that is allowed after the name of an access type in a subtype indication is an index constraint. An access value belongs to a corresponding subtype of an access type either if the access value is the null value or if the value of the designated object satisfies the constraint.

Examples:

```
type ADDRESS is access MEMORY;
type BUFFER_PTR is access TEMP_BUFFER;
```

NOTES

1--An access value delivered by an allocator can be assigned to several variables of the corresponding access type. Hence, it is possible for an object created by an allocator to be designated by more than one variable of the access type. An access value can only designate an object created by an allocator; in particular, it cannot designate an object declared by an object declaration.

2--If the type of the object designated by the access value is an array type, this object is constrained with the array bounds supplied implicitly or explicitly for the corresponding allocator.

3.3.1 Incomplete type declarations

The designated type of an access type can be of any type except a file type (see 3.3). In particular, the type of an element of

the designated type can be another access type or even the same access type. This permits mutually dependent and recursive access types. Declarations of such types require a prior incomplete type declaration for one or more types.

```
incomplete_type_declaration ::=  type identifier ;
```

For each incomplete type declaration there must be a corresponding full type declaration with the same identifier. This full type declaration must occur later and immediately within the same declarative part as the incomplete type declaration to which it corresponds.

Prior to the end of the corresponding full type declaration, the only allowed use of a name that denotes a type declared by an incomplete type declaration is as the type mark in the subtype indication of an access type definition; no constraints are allowed in this subtype indication.

Example of a recursive type:

```
     type CELL;                                        --  An incomplete type
declaration.

     type LINK is access CELL;

     type CELL is
          record
              VALUE    : INTEGER;
              SUCC     : LINK;
              PRED     : LINK;
          end record CELL;
     variable HEAD : LINK := new CELL'(0, null, null);
     variable \NEXT\ : LINK := HEAD.SUCC;
```

Examples of mutually dependent access types:

```
     type PART;                                        --  Incomplete type
declarations.
     type WIRE;

     type PART_PTR is access PART;
     type WIRE_PTR is access WIRE;

     type PART_LIST is array (POSITIVE range <>) of PART_PTR;
     type WIRE_LIST is array (POSITIVE range <>) of WIRE_PTR;

     type PART_LIST_PTR is access PART_LIST;
     type WIRE_LIST_PTR is access WIRE_LIST;

     type PART is
         record
             PART_NAME     : STRING (1 to MAX_STRING_LEN);
             CONNECTIONS   : WIRE_LIST_PTR;
         end record;

     type WIRE is
         record
             WIRE_NAME     : STRING (1 to MAX_STRING_LEN);
             CONNECTS      : PART_LIST_PTR;
         end record;
```

3.3.2 Allocation and deallocation of objects

An object designated by an access value is allocated by an allocator for that type. An allocator is a primary of an expression; allocators are described in 7.3.6 . For each access type, a deallocation operation is implicitly declared immediately following the full type declaration for the type. This deallocation operation makes it possible to deallocate explicitly the storage occupied by a designated object.

Given the following access type declaration:

```
type AT is access T;
```

the following operation is implicitly declared immediately following the access type declaration:

```
procedure DEALLOCATE (P: inout AT) ;
```

Procedure DEALLOCATE takes as its single parameter a variable of the specified access type. If the value of that variable is the null value for the specified access type, then the operation has no effect. If the value of that variable is an access value that designates an object, the storage occupied by that object is returned to the system and may then be reused for subsequent object creation through the invocation of an allocator. The access parameter P is set to the null value for the specified type.

NOTE--If a pointer is copied to a second variable and is then deallocated,the second variable is *not* set to null and thus references invalid storage.

3.4 File types

A file type definition defines a file type. File types are used to define objects representing files in the host system environment. The value of a file object is the sequence of values contained in the host system file.

```
file_type_definition ::=  file of type_mark
```

The type mark in a file type definition defines the subtype of the values contained in the file. The type mark may denote either a constrained or an unconstrained subtype. The base type of this subtype must not be a file type or an access type. If the base type is a composite type, it must not contain a subelement of an access type. If the base type is an array type, it must be a one-dimensional array type.

Examples:

```
     file of STRING           --  Defines a file type that can contain
                              --  an indefinite number of strings of arbitrary
length.
     file of NATURAL          --  Defines a file type that can contain
                              --  only nonnegative integer values.
```

3.4.1 File operations

The language implicitly defines the operations for objects of a file type. Given the following file type declaration:

```
type FT is file of TM;
```

where type mark TM denotes a scalar type, a record type, or a constrained array subtype, the following operations are implicitly declared immediately following the file type declaration:

```
procedure FILE_OPEN (file F: FT;
                     External_Name: in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_OPEN (Status: out FILE_OPEN_STATUS;
                     file F: FT;
                     External_Name: in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_CLOSE (file F: FT);

procedure READ (file F: FT; VALUE: out TM);

procedure WRITE (file F: FT; VALUE: in TM);

function ENDFILE (file F: FT) return BOOLEAN;
```

The FILE_OPEN procedures open an external file specified by the External_Name parameter and associate it with the file object F. If the call to FILE_OPEN is successful (see below), the file object is said to be *open* and the file object has an *access mode* dependent on the value supplied to the Open_Kind parameter (see 14.2).

-- If the value supplied to the Open_Kind parameter is READ_MODE, the access mode of the file object is *read-only*. In addition, the file object is initialized so that a subsequent READ will return the first value in the external file. Values are read from the file object in the order that they appear in the external file.

-- If the value supplied to the Open_Kind parameter is WRITE_MODE, the access mode of the file object is *write-only*. In addition, the external file is made initially empty. Values written to the file object are placed in the external file in the order in which they are written.

-- If the value supplied to the Open_Kind parameter is APPEND_MODE, the access mode of the file object is *write-only*. In addition, the file object is initialized so that values written to it will be added to the end of the external file in the order in which they are written.

In the second form of FILE_OPEN, the value returned through the Status parameter indicates the results of the procedure call:

-- A value of OPEN_OK indicates that the call to FILE_OPEN was successful. If the call to FILE_OPEN specifies an external file that does not exist at the beginning of the call, and if the access mode of the file object passed to the call is write-only, then the external file is created.

-- A value of STATUS_ERROR indicates that the file object already has an external file associated with it.

-- A value of NAME_ERROR indicates that the external file does not exist (in the case of an attempt to read from the external file) or the external file cannot be created (in the case of an attempt to write or append to an external file that does not exist). This value is also returned if the external file cannot be associated with the file object for any reason.

-- A value of MODE_ERROR indicates that the external file cannot be opened with the requested Open_Kind.

The first form of FILE_OPEN causes an error to occur if the second form of FILE_OPEN, when called under identical conditions, would return a Status value other than OPEN_OK.

A call to FILE_OPEN of the first form is *successful* if and only if the call does not cause an error to occur. Similarly, a call to FILE_OPEN of the second form is successful if and only if it returns a Status value of OPEN_OK.

If a file object F is associated with an external file, procedure FILE_CLOSE terminates access to the external file associated with F and closes the external file. If F is not associated with an external file, then FILE_CLOSE has no effect. In either case,

the file object is no longer open after a call to FILE_CLOSE that associates the file object with the formal parameter F.

An implicit call to FILE_CLOSE exists in a subprogram body for every file object declared in the corresponding subprogram declarative part. Each such call associates a unique file object with the formal parameter F and is called whenever the corresponding subprogram completes its execution.

Procedure READ retrieves the next value from a file; it is an error if the access mode of the file object is write-only or if the file object is not open. Procedure WRITE appends a value to a file; it is similarly an error if the access mode of the file object is read-only or if the file is not open. Function ENDFILE returns FALSE if a subsequent READ operation on an open file object whose access mode is read-only can retrieve another value from the file;otherwise, it returns TRUE. Function ENDFILE always returns TRUE for an open file object whose access mode is write-only. It is an error if ENDFILE is called on a file object that is not open.

For a file type declaration in which the type mark denotes an unconstrained array type, the same operations are implicitly declared, except that the READ operation is declared as follows:

```
procedure READ (file F: FT; VALUE: out TM; LENGTH: out Natural);
```

The READ operation for such a type performs the same function as the READ operation for other types, but in addition it returns a value in parameter LENGTH that specifies the actual length of the array value read by the operation. If the object associated with formal parameter VALUE is shorter than this length, then only that portion of the array value read by the operation that can be contained in the object is returned by the READ operation, and the rest of the value is lost. If the object associated with formal parameter VALUE is longer than this length, then the entire value is returned and remaining elements of the object are unaffected by the READ operation.

An error will occur when a READ operation is performed on file F if ENDFILE(F) would return TRUE at that point.

NOTE--Predefined package TEXTIO is provided to support formatted human-readable I/O. It defines type TEXT (a file type representing files of variable-length text strings) and type LINE (an access type that designates such strings). READ and WRITE operations are provided in package TEXTIO that append or extract data from a single line. Additional operations are provided to read or write entire lines and to determine the status of the current line or of the file itself. Package TEXTIO is defined in Section 14.

Section 2

Subprograms and packages

Subprograms define algorithms for computing values or exhibiting behavior. They may be used as computational resources to convert between values of different types, to define the resolution of output values driving a common signal, or to define portions of a process. Packages provide a means of defining these and other resources in a way that allows different design units to share the same declarations.

There are two forms of subprograms: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. Certain functions, designated *pure* functions, return the same value each time they are called with the same values as actual parameters; the remainder,*impure* functions, may return a different value each time they are called, even when multiple calls have the same actual parameter values. In addition, impure functions can update objects outside of their scope and can access a broader class of values than can pure functions. The definition of a subprogram can be given in two parts: a subprogram declaration defining its calling conventions, and a subprogram body defining its execution.

Packages may also be defined in two parts. A package declaration defines the visible contents of a package; a package body provides hidden details. In particular, a package body contains the bodies of any subprograms declared in the package declaration.

2.1 Subprogram declarations

A subprogram declaration declares a procedure or a function, as indicated by the appropriate reserved word.

```
subprogram_declaration ::=
  subprogram_specification ;

subprogram_specification ::=
     procedure designator [ ( formal_parameter_list ) ]
  | [ pure | impure ] function designator [ ( formal_parameter_list ) ]
                     return type_mark

designator ::=  identifier  |  operator_symbol

operator_symbol ::=  string_literal
```

The specification of a procedure specifies its designator and its formal parameters(if any). The specification of a function specifies its designator, its formal parameters (if any), the subtype of the returned value (the *result subtype*), and whether or not the function is pure. A function is *impure* if its specification contains the reserved word **impure**;otherwise, it is said to be *pure*. A procedure designator is always an identifier. A function designator is either an identifier or an operator symbol. A designator that is an operator symbol is used for the overloading of an operator (see 2.3.1). The sequence of characters represented by an operator symbol must be an operator belonging to one of the classes of operators defined in 7.2 . Extra spaces are not allowed in an operator symbol, and the case of letters is not significant.

NOTE--All subprograms can be called recursively.

2.1.1 Formal parameters

The formal parameter list in a subprogram specification defines the formal parameters of the subprogram.

```
formal_parameter_list ::=  parameter_interface_list
```

Formal parameters of subprograms may be constants, variables, signals, or files. In the first three cases, the mode of a parameter determines how a given formal parameter may be accessed within the subprogram. The mode of a formal parameter, together with its class, may also determine how such access is implemented. In the fourth case, that of files, the parameters have no mode.

For those parameters with modes, the only modes that are allowed for formal parameters of a procedure are **in**, **inout**, and **out**. If the mode is **in** and no object class is explicitly specified, **constant** is assumed. If the mode is **inout** or **out**, and no object class is explicitly specified, **variable** is assumed.

For those parameters with modes, the only mode that is allowed for formal parameters of a function is the mode **in** (whether this mode is specified explicitly or implicitly). The object class must be **constant**,**signal**, or **file**. If no object class is explicitly given,**constant** is assumed.

In a subprogram call, the actual designator (see 4.3.2.2) associated with a formal parameter of class **signal** must be a signal. The actual designator associated with a formal of class **variable** must be a variable. The actual designator associated with a formal of class **constant** must be an expression. The actual designator associated with a formal of class **file** must be a file.

NOTE--Attributes of an actual are never passed into a subprogram: references to an attribute of a formal parameter are legal only if that formal has such an attribute. Such references retrieve the value of the attribute associated with the formal.

2.1.1.1 Constant and variable parameters

For parameters of class **constant** or **variable**, only the values of the actual or formal are transferred into or out of the subprogram call. The manner of such transfers, and the accompanying access privileges that are granted for constant and variable parameters, are described in this subclause.

For a nonforeign subprogram having a parameter of a scalar type or an access type, the parameter is passed by copy. At the start of each call, if the mode is **in** or **inout**, the value of the actual parameter is copied into the associated formal parameter; it is an error if, after applying any conversion function or type conversion present in the actual part of the applicable association element (see 4.3.2.2), the value of the actual parameter does not belong to the subtype denoted by the subtype indication of the formal. After completion of the subprogram body, if the mode is **inout** or **out**, the value of the formal parameter is copied back into the associated actual parameter; it is similarly an error if, after applying any conversion function or type conversion present in the formal part of the applicable association element, the value of the formal parameter does not belong to the subtype denoted by the subtype indication of the actual.

For a nonforeign subprogram having a parameter whose type is an array or record, an implementation may pass parameter values by copy, as for scalar types. If a parameter of mode **out** is passed by copy, then the range of each index position of the actual parameter is copied in, and likewise for its subelements or slices. Alternatively, an implementation may achieve these effects by reference; that is, by arranging that every use of the formal parameter (to read or update its value) be treated as a use of the associated actual parameter throughout the execution of the subprogram call. The language does not define which of these two mechanisms is to be adopted for parameter passing, nor whether different calls to the same subprogram are to use the same mechanism. The execution of a subprogram is erroneous if its effect depends on which mechanism is selected by the implementation.

For a formal parameter of a constrained array subtype of mode **in** or **inout**, it is an error if the value of the associated actual parameter(after application of any conversion function or type conversion present in the actual part) does not contain a matching element for each element of the formal. For a formal parameter whose declaration contains a subtype indication denoting an unconstrained array type, the subtype of the formal in any call to the subprogram is taken from the actual

associated with that formal in the call to the subprogram. It is also an error if, in either case, the value of each element of the actual array (after applying any conversion function or type conversion present in the actual part) does not belong to the element subtype of the formal. If the formal parameter is of mode **out** or **inout**,it is also an error if, at the end of the subprogram call, the value of each element of the formal (after applying any conversion function or type conversion present in the formal part) does not belong to the element subtype of the actual.

NOTES

1--For parameters of array and record types, the parameter-passing rules imply that if no actual parameter of such a type is accessible by more than one path, then the effect of a subprogram call is the same whether or not the implementation uses copying for parameter passing. If, however, there are multiple access paths to such a parameter (for example, if another formal parameter is associated with the same actual parameter), then the value of the formal is undefined after updating the actual other than by updating the formal. A description using such an undefined value is erroneous.

2--As a consequence of the parameter-passing conventions for variables, if a procedure is called with a shared variable (see 4.3.1.3) as an actual to a formal variable parameter of modes **inout** or **out**, the shared variable may not be updated until the procedure completes its execution. Furthermore, a formal variable parameter with modes **in** or **inout** may not reflect updates made to a shared variable associated with it as an actual during the execution of the subprogram, including updates made to the actual during the execution of a wait statement within a procedure.

2.1.1.2 Signal parameters

For a formal parameter of class **signal**, references to the signal, the driver of the signal, or both, are passed into the subprogram call.

For a signal parameter of mode **in** or **inout**, the actual signal is associated with the corresponding formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, a reference to the formal signal parameter within an expression is equivalent to a reference to the actual signal.

It is an error if signal-valued attributes 'STABLE, 'QUIET, 'TRANSACTION, and 'DELAYED of formal signal parameters of any mode are read within a subprogram.

A process statement contains a driver for each actual signal associated with a formal signal parameter of mode **out** or **inout** in a subprogram call. Similarly, a subprogram contains a driver for each formal signal parameter of mode **out** or **inout** declared in its subprogram specification.

For a signal parameter of mode **inout** or **out**, the driver of an actual signal is associated with the corresponding driver of the formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, an assignment to the driver of a formal signal parameter is equivalent to an assignment to the driver of the actual signal.

If an actual signal is associated with a signal parameter of any mode, the actual must be denoted by a static signal name. It is an error if a conversion function or type conversion appears in either the formal part or the actual part of an association element that associates an actual signal with a formal signal parameter.

If an actual signal is associated with a signal parameter of any mode, and if the type of the formal is a scalar type, then it is an error if the bounds and direction of the subtype denoted by the subtype indication of the formal are not identical to the bounds and direction of the subtype denoted by the subtype indication of the actual.

If an actual signal is associated with a formal signal parameter, and if the formal is of a constrained array subtype, then it is an error if the actual does not contain a matching element for each element of the formal. If an actual signal is associated with a formal signal parameter, and if the subtype denoted by the subtype indication of the declaration of the formal is an unconstrained array type, then the subtype of the formal in any call to the subprogram is taken from the actual associated with that formal in the call to the subprogram. It is also an error if the mode of the formal is **in** or **inout** and if the value of each element of the actual array does not belong to the element subtype of the formal.

A formal signal parameter is a guarded signal if and only if it is associated with an actual signal that is a guarded signal. It is an error if the declaration of a formal signal parameter includes the reserved word **bus**(see 4.3.2).

NOTE--It is a consequence of the preceding rules that a procedure with an **out** or **inout** signal parameter called by a process does not have to complete in order for any assignments to that signal parameter within the procedure to take effect. Assignments to the driver of a formal signal parameter are equivalent to assignments directly to the actual driver contained in the process calling the procedure.

2.1.1.3 File parameters

For parameters of class **file**, references to the actual file are passed into the subprogram. No particular parameter-passing mechanism is defined by the language, but a reference to the formal parameter must be equivalent to a reference to the actual parameter. It is an error if an association element associates an actual with a formal parameter of a file type and that association element contains a conversion function or type conversion. It is also an error if a formal of a file type is associated with an actual that is not of a file type.

At the beginning of a given subprogram call, a file parameter is open (see 3.4.1) if and only if the actual file object associated with the given parameter in a given subprogram call is also open. Similarly, at the beginning of a given subprogram call, both the access mode of and external file associated with (see 3.4.1) an open file parameter are the same as, respectively, the access mode of and the external file associated with the actual file object associated with the given parameter in the subprogram call.

At the completion of the execution of a given subprogram call, the actual file object associated with a given file parameter is open if and only if the formal parameter is also open. Similarly, at the completion of the execution of a given subprogram call, the access mode of and the external file associated with an open actual file object associated with a given file parameter are the same as, respectively, the access mode of and the external file associated with the associated formal parameter.

2.2 Subprogram bodies

A subprogram body specifies the execution of a subprogram.

```
subprogram_body ::=
  subprogram_specification is
          subprogram_declarative_part
  begin
          subprogram_statement_part
  end [ subprogram_kind ] [ designator ] ;

subprogram_declarative_part ::=
  { subprogram_declarative_item }

subprogram_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
```

```
    | use_clause
    | group_template_declaration
    | group_declaration

subprogram_statement_part ::=
  { sequential_statement }

subprogram_kind ::=  procedure | function
```

The declaration of a subprogram is optional. In the absence of such a declaration,the subprogram specification of the subprogram body acts as the declaration. For each subprogram declaration, there must be a corresponding body. If both a declaration and a body are given, the subprogram specification of the body must conform (see 2.7) to the subprogram specification of the declaration. Furthermore, both the declaration and the body must occur immediately within the same declarative region (see 10.1).

If a subprogram kind appears at the end of a subprogram body, it must repeat the reserved word given in the subprogram specification. If a designator appears at the end of a subprogram body, it must repeat the designator of the subprogram.

It is an error if a variable declaration in a subprogram declarative part declares a shared variable. (See 4.3.1.3 and 8.1.4 .)

A *foreign subprogram* is one that is decorated with the attribute 'FOREIGN, defined in package STANDARD (see 14.2). The STRING value of the attribute may specify implementation-dependent information about the foreign subprogram. Foreign subprograms may have non-VHDL implementations. An implementation may place restrictions on the allowable modes, classes, and types of the formal parameters to a foreign subprogram; such restrictions may include restrictions on the number and allowable order of the parameters.

Excepting foreign subprograms, the algorithm performed by a subprogram is defined by the sequence of statements that appears in the subprogram statement part. For a foreign subprogram, the algorithm performed is implementation defined.

The execution of a subprogram body is invoked by a subprogram call. For this execution, after establishing the association between the formal and actual parameters, the sequence of statements of the body is executed if the subprogram is not a foreign subprogram; otherwise, an implementation-defined action occurs. Upon completion of the body or implementation-dependent action, return is made to the caller (and any necessary copying back of formal to actual parameters occurs).

A process or a subprogram is said to be a *parent* of a given subprogram S if that process or subprogram contains a procedure call or function call for S or for a parent of S.

An *explicit signal* is a signal other than an implicit signal GUARD or other than one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION. The *explicit ancestor* of an implicit signal is found as follows. The implicit signal GUARD has no explicit ancestor. An explicit ancestor of an implicit signal defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION is the signal found by recursively examining the prefix of the attribute. If the prefix denotes an explicit signal, or a member (see Section 3) of an explicit signal then that is the explicit ancestor of the implicit signal. Otherwise, if the prefix is one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE,'QUIET, or 'TRANSACTION, this rule is recursively applied. If the prefix is an implicit signal GUARD, then the signal has no explicit ancestor.

If a pure function subprogram is a parent of a given procedure and if that procedure contains a reference to an explicitly declared signal or variable object, or a slice or subelement (or slice thereof), of an explicit signal, then that object must be declared within the declarative region formed by the function (see 10.1) or within the declarative region formed by the procedure; this rule also holds for the explicit ancestor, if any, of an implicit signal and also for the implicit signal GUARD. If a pure function is the parent of a given procedure, then that procedure must not contain a reference to an explicitly declared file object(see 4.3.1.4) or to a shared variable (see 4.3.1.3).

Similarly, if a pure function subprogram contains a reference to an explicitly declared signal or variable object, or a slice or

subelement (or slice thereof) of an explicit signal , then that object must be declared within the declarative region formed by the function; this rule also holds for the explicit ancestor, if any,of an implicit signal and also for the implicit signal GUARD. A pure function must not contain a reference to an explicitly declared file object.

A pure function must not be the parent of an impure function.

The rules of the preceding four paragraphs apply to all pure function subprograms. For pure functions that are not foreign subprograms, violations of any of these rules are errors. However, since implementations cannot in general check that such rules hold for pure function subprograms that are foreign subprograms, a description calling pure foreign function subprograms not adhering to these rules is erroneous.

Example:

```
-- The declaration of a foreign function subprogram:

     package P is
        function F return INTEGER;
        attribute FOREIGN of F: function is "implementation-dependent information";
     end package P;
```

NOTES

1--It follows from the visibility rules that a subprogram declaration must be given if a call of the subprogram occurs textually before the subprogram body, and that such a declaration must occur before the call itself.

2--The preceding rules concerning pure function subprograms, together with the fact that function parameters may only be of mode **in**, imply that a pure function has no effect other than the computation of the returned value. Thus, a pure function invoked explicitly as part of the elaboration of a declaration, or one invoked implicitly as part of the simulation cycle, is guaranteed to have no effect on other objects in the description.

3--VHDL does not define the parameter-passing mechanisms for foreign subprograms.

4--The declarative parts and statement parts of subprograms decorated with the 'FOREIGN attribute are subject to special elaboration rules. See 12.3 and 12.4 .

5--A pure function subprogram may not reference a shared variable. This prohibition exists because a shared variable may not be declared in a subprogram declarative part and a pure function may not reference any variable declared outside of its declarative region.

2.3 Subprogram overloading

Two formal parameter lists are said to have the same *parameter type profile* if and only if they have the same number of parameters, and if at each parameter position the corresponding parameters have the same base type. Two subprograms are said to have the same *parameter and result type profile* if and only if both have the same parameter type profile, and if either both are functions with the same result base type or neither of the two is a function.

A given subprogram designator can be used in several subprogram specifications. The subprogram designator is then said to be overloaded; the designated subprograms are also said to be overloaded and to overload each other. If two subprograms overload each other, one of them can hide the other only if both subprograms have the same parameter and result type profile.

A call to an overloaded subprogram is ambiguous (and therefore is an error) if the name of the subprogram, the number of parameter associations, the types and order of the actual parameters, the names of the formal parameters (if named

associations are used), and the result type (for functions) are not sufficient to identify exactly one (overloaded) subprogram specification.

Similarly, a reference to an overloaded resolution function name in a subtype indication is ambiguous (and is therefore an error) if the name of the function, the number of formal parameters, the result type, and the relationships between the result type and the types of the formal parameters (as defined in 2.4) are not sufficient to identify exactly one (overloaded) subprogram specification.

Examples:

```
--  Declarations of overloaded subprograms:

     procedure Dump(F: inout Text; Value: Integer);
     procedure Dump(F: inout Text; Value: String);

     procedure Check (Setup: Time; signal D: Data; signal C: Clock);
     procedure Check (Hold: Time; signal C: Clock; signal D: Data);

--  Calls to overloaded subprograms:

     Dump (Sys_Output, 12) ;
     Dump (Sys_Error, "Actual output does not match expected output") ;

     Check (Setup=>10 ns, D=>DataBus, C=>Clk1) ;
     Check (Hold=>5 ns, D=>DataBus, C=>Clk2);
     Check (15 ns, DataBus, Clk) ;
      --  Ambiguous if the base type of DataBus is the same type as the base type of
Clk.
```

NOTES

1--The notion of parameter and result type profile does not include parameter names, parameter classes, parameter modes, parameter subtypes, or default expressions or their presence or absence.

2--Ambiguities may (but need not) arise when actual parameters of the call of an overloaded subprogram are themselves overloaded function calls, literals, or aggregates. Ambiguities may also (but need not) arise when several overloaded subprograms belonging to different packages are visible. These ambiguities can usually be solved in two ways: qualified expressions can be used for some or all actual parameters and for the result, if any; or the name of the subprogram can be expressed more explicitly as an expanded name (see 6.3).

2.3.1 Operator overloading

The declaration of a function whose designator is an operator symbol is used to overload an operator. The sequence of characters of the operator symbol must be one of the operators in the operator classes defined in 7.2 .

The subprogram specification of a unary operator must have a single parameter. The subprogram specification of a binary operator must have two parameters; for each use of this operator, the first parameter is associated with the left operand, and the second parameter is associated with the right operand.

For each of the operators "+" and "-", overloading is allowed both as a unary operator and as a binary operator.

NOTES

1--Overloading of the equality operator does not affect the selection of choices in a case statement or in a selected signal

assignment statement; nor does it have an affect on the propagation of signal values.

2--A user-defined operator that has the same designator as a short-circuit operator (that is, that overloads the short-circuit operator) is not invoked in a short-circuit manner. Specifically, calls to the user-defined operator always evaluate both arguments prior to the execution of the function.

3--Functions that overload operator symbols may also be called using function call notation rather than operator notation. This statement is also true of the predefined operators themselves.

Examples:

```
type MVL is ('0', '1', 'Z', 'X') ;

function "and" (Left, Right: MVL) return MVL ;
function "or" (Left, Right: MVL) return MVL ;
function "not" (Value: MVL) return MVL ;

signal Q,R,S: MVL ;

Q <= 'X' or '1';
R <= "or" ('0','Z');
S <= (Q and R) or not S;
```

2.3.2 Signatures

A signature distinguishes between overloaded subprograms and overloaded enumeration literals based on their parameter and result type profiles. A signature can be used in an attribute name, entity designator, or alias declaration.

```
signature ::=  [ [ type_mark { , type_mark } ] [ return type_mark ] ]
```

(Note that the initial and terminal brackets are part of the syntax of signatures and do not indicate that the entire right-hand side of the production is optional.) A signature is said to *match* the parameter and result type profile of a given subprogram if and only if all of the following conditions hold:

-- The number of type marks prior to the reserved word **return**, if any, matches the number of formal parameters of the subprogram

-- At each parameter position, the base type denoted by the type mark of the signature is the same as the base type of the corresponding formal parameter of the subprogram

-- If the reserved word **return** is present, the subprogram is a function and the base type of the type mark following the reserved word in the signature is the same as the base type of the return type of the function, or the reserved word **return** is absent and the subprogram is a procedure

Similarly, a signature is said to match the parameter and result type profile of a given enumeration literal if the signature matches the parameter and result type profile of the subprogram equivalent to the enumeration literal, defined in 3.1.1 .

Example:

```
attribute BuiltIn of "or" [MVL, MVL return MVL]: function is TRUE;
  --  Because of the presence of the signature, this attribute specification
  --  decorates only the "or" function defined in the previous section.

attribute Mapping of JMP [return OpCode] : literal is "001";
```

2.4 Resolution functions

A resolution function is a function that defines how the values of multiple sources of a given signal are to be resolved into a single value for that signal. Resolution functions are associated with signals that require resolution by including the name of the resolution function in the declaration of the signal or in the declaration of the subtype of the signal. A signal with an associated resolution function is called a resolved signal (see 4.3.1.2).

A resolution function must be a pure function (see 2.1); moreover, it must have a single input parameter of class **constant** that is a one-dimensional,unconstrained array whose element type is that of the resolved signal. The type of the return value of the function must also be that of the signal. Errors occur at the place of the subtype indication containing the name of the resolution function if any of these checks fail (see 4.2).

The resolution function associated with a resolved signal determines the *resolved value* of the signal as a function of the collection of inputs from its multiple sources. If a resolved signal is of a composite type, and if subelements of that type also have associated resolution functions, such resolution functions have no effect on the process of determining the resolved value of the signal. It is an error if a resolved signal has more connected sources than the number of elements in the index type of the unconstrained array type used to define the parameter of the corresponding resolution function.

Resolution functions are implicitly invoked during each simulation cycle in which corresponding resolved signals are active (see 12.6.1). Each time a resolution function is invoked, it is passed an array value, each element of which is determined by a corresponding source of the resolved signal, but excluding those sources that are drivers whose values are determined by null transactions (see 8.4.1). Such drivers are said to be *off*. For certain invocations (specifically, those involving the resolution of sources of a signal declared with the signal kind **bus**), a resolution function may thus be invoked with an input parameter that is a null array; this occurs when all sources of the bus are drivers, and they are all off. In such a case, the resolution function returns a value representing the value of the bus when no source is driving it.

Example:

```
function WIRED_OR (Inputs: BIT_VECTOR) return BIT is
  constant FloatValue: BIT := '0';
begin
  if Inputs'Length = 0 then
            --  This is a bus whose drivers are all off.
            return FloatValue;
  else
            for I in Inputs'Range loop
                      if Inputs(I) = '1' then
                                Return '1';
                      end If;
            end Loop;
            Return '0';
  end if;
end function WIRED_OR;
```

2.5 Package declarations

A package declaration defines the interface to a package. The scope of a declaration within a package can be extended to other design units.

```
package_declaration ::=
  package identifier is
```

```
            package_declarative_part
    end [ package ] [ package_simple_name ] ;

  package_declarative_part ::=
    { package_declarative_item }

  package_declarative_item ::=
     subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
```

If a simple name appears at the end of the package declaration, it must repeat the identifier of the package declaration.

Items declared immediately within a package declaration become visible by selection within a given design unit wherever the name of that package is visible in the given unit. Such items may also be made directly visible by an appropriate use clause (see 10.4).

NOTE--Not all packages will have a package body. In particular, a package body is unnecessary if no subprograms or deferred constants are declared in the package declaration.

Examples:

```
--  A package declaration that needs no package body:

    package TimeConstants is
      constant tPLH :  Time := 10 ns;
      constant tPHL :  Time := 12 ns;
      constant tPLZ :  Time := 7 ns;
      constant tPZL :  Time := 8 ns;
      constant tPHZ :  Time := 8 ns;
      constant tPZH :  Time := 9 ns;
    end TimeConstants ;

--  A package declaration that needs a package body:

    package TriState is
      type Tri is ('0', '1', 'Z', 'E');
      function BitVal (Value: Tri) return Bit ;
      function TriVal (Value: Bit) return Tri;
      type TriVector is array (Natural range <>) of Tri ;
      function Resolve (Sources: TriVector) return Tri ;
    end package TriState ;
```

2.6 Package bodies

A package body defines the bodies of subprograms and the values of deferred constants declared in the interface to the package.

```
package_body ::=
  package body package_simple_name is
          package_body_declarative_part
  end [ package body ] [ package_simple_name ] ;

package_body_declarative_part ::=
  { package_body_declarative_item }

package_body_declarative_item ::=
   subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | shared_variable_declaration
  | file_declaration
  | alias_declaration
  | use_clause
  | group_template_declaration
  | group_declaration
```

The simple name at the start of a package body must repeat the package identifier. If a simple name appears at the end of the package body, it must be the same as the identifier in the package declaration.

In addition to subprogram body and constant declarative items, a package body may contain certain other declarative items to facilitate the definition of the bodies of subprograms declared in the interface. Items declared in the body of a package cannot be made visible outside of the package body.

If a given package declaration contains a deferred constant declaration (see 4.3.1.1), then a constant declaration with the same identifier must appear as a declarative item in the corresponding package body. This object declaration is called the *full* declaration of the deferred constant. The subtype indication given in the full declaration must conform to that given in the deferred constant declaration.

Within a package declaration that contains the declaration of a deferred constant, and within the body of that package (before the end of the corresponding full declaration), the use of a name that denotes the deferred constant is only allowed in the default expression for a local generic, local port, or formal parameter. The result of evaluating an expression that references a deferred constant before the elaboration of the corresponding full declaration is not defined by the language.

Example:

```
package body TriState is

  function BitVal (Value: Tri) return Bit is
          constant Bits : Bit_Vector := "0100";
  begin
          return Bits(Tri'Pos(Value));
  end;

  function TriVal (Value: Bit) return Tri is
```

```
    begin
            return Tri'Val(Bit'Pos(Value));
    end;

    function Resolve (Sources: TriVector) return Tri is
            variable V: Tri := 'Z';
    begin
            for i in Sources'Range loop
                    if Sources(i) /= 'Z' then
                            if V = 'Z' then
                                    V := Sources(i);
                            else
                                    return 'E';
                            end if;
                    end if;
            end loop;
            return V;
    end;
end package body TriState ;
```

2.7 Conformance rules

Whenever the language rules either require or allow the specification of a given subprogram to be provided in more than one place, the following variations are allowed at each place:

-- A numeric literal can be replaced by a different numeric literal if and only if both have the same value.

-- A simple name can be replaced by an expanded name in which this simple name is the selector if and only if at both places the meaning of the simple name is given by the same declaration.

Two subprogram specifications are said to *conform* if, apart from comments and the above allowed variations, both specifications are formed by the same sequence of lexical elements and if corresponding lexical elements are given the same meaning by the visibility rules.

Conformance is likewise defined for subtype indications in deferred constant declarations.

1--A simple name can be replaced by an expanded name even if the simple name is itself the prefix of a selected name. For example, Q.R can be replaced by P.Q.R if Q is declared immediately within P.

2--The subprogram specification of an impure function is never conformant to a subprogram specification of a pure function.

3--The following specifications do not conform since they are not formed by the same sequence of lexical elements:

```
procedure P (X,Y : INTEGER)
procedure P (X: INTEGER; Y : INTEGER)
procedure P (X,Y : in INTEGER)
```

Section 1

Design entites and configurations

The *design entity* is the primary hardware abstraction in VHDL. It represents a portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function. A design entity may represent an entire system, a subsystem, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. A *configuration* can be used to describe how design entities are put together to form a complete design.

A design entity may be described in terms of a hierarchy of *blocks*, each of which represents a portion of the whole design. The top-level block in such a hierarchy is the design entity itself; such a block is an *external* block that resides in a library and may be used as a component of other designs. Nested blocks in the hierarchy are *internal* blocks, defined by block statements (see 9.1).

A design entity may also be described in terms of interconnected components. Each component of a design entity may be bound to a lower-level design entity in order to define the structure or behavior of that component. Successive decomposition of a design entity into components, and binding those components to other design entities that may be decomposed in like manner, results in a hierarchy of design entities representing a complete design. Such a collection of design entities is called a *design hierarchy*. The bindings necessary to identify a design hierarchy can be specified in a configuration of the top-level entity in the hierarchy.

This section describes the way in which design entities and configurations are defined. A design entity is defined by an *entity declaration* together with a corresponding *architecture body*. A configuration is defined by a *configuration declaration*.

1.1 Entity declarations

An entity declaration defines the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration maybe shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each with the same interface.

```
entity_declaration ::=
     entity identifier is
          entity_header
          entity_declarative_part
  [   begin
          entity_statement_part ]
     end [ entity ] [ entity_simple_name ] ;
```

The entity header and entity declarative part consist of declarative items that pertain to each design entity whose interface is defined by the entity declaration. The entity statement part, if present, consists of concurrent statements that are present in each such design entity.

If a simple name appears at the end of an entity declaration, it must repeat the identifier of the entity declaration.

1.1.1 Entity header

The entity header declares objects used for communication between a design entity and its environment.

```
entity_header ::=
     [ formal_generic_clause ]
     [ formal_port_clause ]

generic_clause ::=
    generic ( generic_list ) ;

port_clause ::=
    port ( port_list ) ;
```

The generic list in the formal generic clause defines generic constants whose values may be determined by the environment. The port list in the formal port clause defines the input and output ports of the design entity.

In certain circumstances, the names of generic constants and ports declared in the entity header become visible outside of the design entity (see 10.2 and 10.3).

Examples:

```
--An entity declaration with port declarations only:

     entity Full_Adder is
          port (X, Y, Cin: in Bit; Cout, Sum: out Bit) ;
     end Full_Adder ;

--An entity declaration with generic declarations also:

     entity AndGate is
          generic
              (N: Natural := 2);
          port
             (Inputs: in       Bit_Vector (1 to N);
              Result: out      Bit) ;
      end entity AndGate ;

--An entity declaration with neither:

      entity TestBench is
      end TestBench ;
```

1.1.1.1 Generics

Generics provide a channel for static information to be communicated to a block from its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements.

```
generic_list ::=  generic_interface_list
```

The generics of a block are defined by a generic interface list; interface lists are described in 4.3.2.1 . Each interface element in such a generic interface list declares a formal generic.

The value of a generic constant may be specified by the corresponding actual in a generic association list. If no such actual is specified for a given formal generic (either because the formal generic is unassociated or because the actual is **open**), and if a default expression is specified for that generic, the value of this expression is the value of the generic. It is an error if no actual is specified for a given formal generic and no default expression is present in the corresponding interface element. It is an error if some of the subelements of a composite formal generic are connected and others are either unconnected or unassociated.

NOTE--Generics may be used to control structural, dataflow, or behavioral characteristics of a block, or may simply be used as documentation. In particular, generics may be used to specify the size of ports; the number of subcomponents within a block; the timing characteristics of a block; or even the physical characteristics of a design such as temperature, capacitance,location, etc.

1.1.1.2 Ports

Ports provide channels for dynamic communication between a block and its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements, including those equivalent to component instantiation statements and generate statements (see 9.7).

```
port_list ::=  port_interface_list
```

The ports of a block are defined by a port interface list; interface lists are described in 4.3.2.1 . Each interface element in the port interface list declares a formal port.

To communicate with other blocks, the ports of a block can be associated with signals in the environment in which the block is used. Moreover, the ports of a block may be associated with an expression in order to provide these ports with constant driving values; such ports must be of mode **in**. A port is itself a signal (see 4.3.1.2); thus, a formal port of a block may be associated as an actual with a formal port of an inner block. The port, signal, or expression associated with a given formal port is called the *actual* corresponding to the formal port (see 4.3.2.2). The actual, if a port or signal, must be denoted by a static name (see 6.1). The actual, if an expression, must be a globally static expression (see 7.4).

After a given description is completely elaborated (see Section 12), if a formal port is associated with an actual that is itself a port, then the following restrictions apply depending upon the mode (see 4.3.2) of the formal port:

a. For a formal port of mode **in**, the associated actual may only be a port of mode **in**, **inout**,or **buffer**.

b. For a formal port of mode **out**, the associated actual may only be a port of mode **out** or **inout**.

c. For a formal port of mode **inout**, the associated actual may only be a port of mode **inout**.

d. For a formal port of mode **buffer**, the associated actual may only be a port of mode **buffer**.

e. For a formal port of mode **linkage**, the associated actual may be a port of any mode.

A **buffer** port may have at most one source (see 4.3.1.2 and 4.3.2).Furthermore, after a description is completely elaborated (see Section 12), any actual associated with a formal buffer port may have at most one source.

If a formal port is associated with an actual port, signal, or expression, then the formal port is said to be *connected*. If a formal port is instead associated with the reserved word **open**, then the formal is said to be *unconnected*. A port of mode **in** may be unconnected or unassociated (see 4.3.2.2) only if its declaration includes a default expression (see 4.3.2). A port of any mode other than **in** may be unconnected or unassociated as long as its type is not an unconstrained array type. It is an error if some of the subelements of a composite formal port are connected and others are either unconnected or unassociated.

1.1.2 Entity declarative part

The entity declarative part of a given entity declaration declares items that are common to all design entities whose interfaces are defined by the given entity declaration.

```
entity_declarative_part ::=
     { entity_declarative_item }

entity_declarative_item ::=
        subprogram_declaration
     | subprogram_body
     | type_declaration
     | subtype_declaration
     | constant_declaration
     | signal_declaration
     | shared_variable_declaration
     | file_declaration
     | alias_declaration
     | attribute_declaration
     | attribute_specification
     | disconnection_specification
     | use_clause
     | group_template_declaration
     | group_declaration
```

Names declared by declarative items in the entity declarative part of a given entity declaration are visible within the bodies of corresponding design entities, as well as within certain portions of a corresponding configuration declaration.

Example:

```
--An entity declaration with entity declarative items:

     entity ROM is
          port (     Addr:  in   Word;
                     Data:  out  Word;
                     Sel:   in   Bit);
          type       Instruction is array (1 to 5) of Natural;
          type       Program is array (Natural range <>) of Instruction;
          use        Work.OpCodes.all, Work.RegisterNames.all;
          constant ROM_Code: Program :=
              (


                        (STM,  R14,  R12,  12,  R13) ,
```

```
                    (LD,    R7,   32,   0,  R1 ) ,
                    (BAL,  R14,    0,   0,  R7 ) ,
                           .
                           .    --  etc.
                           .
            );
    end ROM;
```

NOTE--The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 12.3 .

1.1.3 Entity statement part

The entity statement part contains concurrent statements that are common to each design entity with this interface.

```
    entity_statement_part ::=
         { entity_statement }

    entity_statement ::=
            concurrent_assertion_statement
         | passive_concurrent_procedure_call
         | passive_process_statement
```

Only concurrent assertion statements, concurrent procedure call statements, or process statements may appear in the entity statement part. All such statements must be passive (see 9.2). Such statements may be used to monitor the operating conditions or characteristics of a design entity.

```
--An entity declaration with statements:

         entity Latch is
              port (Din:    in     Word;
                    Dout:   out     Word;
                    Load:   in      Bit;
                    Clk:    in      Bit );
              constant Setup: Time := 12 ns;
              constant PulseWidth: Time := 50 ns;
              use Work.TimingMonitors.all;
         begin
              assert Clk='1' or Clk'Delayed'Stable (PulseWidth);
              CheckTiming (Setup, Din, Load, Clk);
         end ;
```

NOTE--The entity statement part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 12.4 .

1.2 Architecture bodies

An architecture body defines the body of a design entity. It specifies the relationships between the inputs and outputs of a design entity and may be expressed in terms of structure, dataflow, or behavior. Such specifications may be partial or complete.

```
architecture_body ::=
    architecture identifier of entity_name is
        architecture_declarative_part
    begin
        architecture_statement_part
    end [ architecture ] [ architecture_simple_name ] ;
```

The identifier defines the simple name of the architecture body; this simple name distinguishes architecture bodies associated with the same entity declaration.

The entity name identifies the name of the entity declaration that defines the interface of this design entity. For a given design entity, both the entity declaration and the associated architecture body must reside in the same library.

If a simple name appears at the end of an architecture body, it must repeat the identifier of the architecture body.

More than one architecture body may exist corresponding to a given entity declaration. Each declares a different body with the same interface; thus,each together with the entity declaration represents a different design entity with the same interface.

NOTE--Two architecture bodies that are associated with different entity declarations may have the same simple name, even if both architecture bodies(and the corresponding entity declarations) reside in the same library.

1.2.1 Architecture declarative part

The architecture declarative part contains declarations of items that are available for use within the block defined by the design entity.

```
architecture_declarative_part ::=
    { block_declarative_item }

block_declarative_item ::=
       subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | configuration_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
```

The various kinds of declaration are described in Section 4, and the various kinds of specification are described in Section 5. The use clause, which makes externally defined names visible within the block, is described in Section 10.

NOTE--The declarative part of an architecture decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 12.3 .

1.2.2 Architecture statement part

The architecture statement part contains statements that describe the internal organization and/or operation of the block defined by the design entity.

```
architecture_statement_part ::=
     { concurrent_statement }
```

All of the statements in the architecture statement part are concurrent statements,which execute asynchronously with respect to one another. The various kinds of concurrent statements are described in Section 9.

Examples:

```
--A body of entity Full_Adder:

     architecture DataFlow of Full_Adder is
         signal A,B: Bit;
     begin
         A <= X xor Y;
         B <= A and Cin;
         Sum <= A xor Cin;
         Cout <= B or (X and Y);
     end architecture DataFlow ;

--A body of entity TestBench:

     library Test;
     use Test.Components.all;
     architecture Structure of TestBench is
         component Full_Adder
            port (X, Y, Cin: Bit; Cout, Sum: out Bit);
         end component;

         signal A,B,C,D,E,F,G: Bit;
         signal OK: Boolean;
     begin
         UUT:         Full_Adder         port map (A,B,C,D,E);
         Generator:   AdderTest          port map (A,B,C,F,G);
         Comparator:  AdderCheck         port map (D,E,F,G,OK);
     end Structure;

--A body of entity AndGate:

     architecture Behavior of AndGate is
     begin
         process (Inputs)
             variable Temp: Bit;
         begin
             Temp := '1';
```

```
            for i in Inputs'Range loop
                 if Inputs(i) = '0' then
                       Temp := '0';
                       exit;
                 end if;
            end loop;
            Result <= Temp after 10 ns;
        end process;
    end Behavior;
```

NOTE--The statement part of an architecture decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 12.4 .

1.3 Configuration declarations

The binding of component instances to design entities is performed by configuration specifications (see 5.2); such specifications appear in the declarative part of the block in which the corresponding component instances are created. In certain cases, however, it may be appropriate to leave unspecified the binding of component instances in a given block and to defer such specification until later. A configuration declaration provides the mechanism for specifying such deferred bindings.

```
configuration_declaration ::=
    configuration identifier of entity_name is
        configuration_declarative_part
        block_configuration
    end [ configuration ] [ configuration_simple_name ] ;

configuration_declarative_part ::=
    { configuration_declarative_item }

configuration_declarative_item ::=
      use_clause
    | attribute_specification
    | group_declaration
```

The entity name identifies the name of the entity declaration that defines the design entity at the apex of the design hierarchy. For a configuration of a given design entity, both the configuration declaration and the corresponding entity declaration must reside in the same library.

If a simple name appears at the end of a configuration declaration, it must repeat the identifier of the configuration declaration.

NOTES

1--A configuration declaration achieves its effect entirely through elaboration (see Section 12). There are no behavioral semantics associated with a configuration declaration.

2--A given configuration may be used in the definition of another, more complex configuration.

Examples:

```
--An architecture of a microprocessor:

     architecture Structure_View of Processor is
         component ALU port ( ··· );  end component;
         component MUX port ( ··· );  end component;
         component Latch port ( ··· );  end component;
     begin
         A1: ALU port map ( ··· ) ;
         M1: MUX port map ( ··· ) ;
         M2: MUX port map ( ··· ) ;
         M3: MUX port map ( ··· ) ;
         L1: Latch port map ( ··· ) ;
         L2: Latch port map ( ··· ) ;
     end Structure_View ;

--A configuration of the microprocessor:

     library TTL, Work ;
     configuration V4_27_87 of Processor is
         use Work.all ;
         for Structure_View
             for A1: ALU
                 use configuration TTL.SN74LS181 ;
             end for ;
             for M1,M2,M3: MUX
                 use entity Multiplex4 (Behavior) ;
             end for ;
             for all: Latch
                    -- use defaults
             end for ;
         end for ;
     end configuration V4_27_87 ;
```

1.3.1 Block configuration

A block configuration defines the configuration of a block. Such a block maybe either an internal block defined by a block statement or an external block defined by a design entity. If the block is an internal block, the defining block statement may be either an explicit block statement or an implicit block statement that is itself defined by a generate statement.

```
     block_configuration ::=
         for block_specification
              { use_clause }
              { configuration_item }
         end for ;

     block_specification ::=
           architecture_name
         | block_statement_label
         | generate_statement_label [ ( index_specification ) ]

     index_specification ::=
           discrete_range
```

```
      | static_expression

  configuration_item ::=
        block_configuration
      | component_configuration
```

The block specification identifies the internal or external block to which this block configuration applies.

If a block configuration appears immediately within a configuration declaration, then the block specification of that block configuration must bean architecture name, and that architecture name must denote a design entity body whose interface is defined by the entity declaration denoted by the entity name of the enclosing configuration declaration.

If a block configuration appears immediately within a component configuration,then the corresponding components must be fully bound (see 5.2.1.1), the block specification of that block configuration must be an architecture name, and that architecture name must denote the same architecture body as that to which the corresponding components are bound.

If a block configuration appears immediately within another block configuration, then the block specification of the contained block configuration must be a block statement or generate statement label, and the label must denote a block statement or generate statement that is contained immediately within the block denoted by the block specification of the containing block configuration.

If the scope of a declaration (see 10.2) includes the end of the declarative part of a block corresponding to a given block configuration, then the scope of that declaration extends to each configuration item contained in that block configuration, with the exception of block configurations that configure external blocks. In addition, if a declaration is visible (either directly or by selection) at the end of the declarative part of a block corresponding to a given block configuration, then the declaration is visible in each configuration item contained in that block configuration, with the exception of block configurations that configure external blocks. Additionally, if a given declaration is a homograph of a declaration that a use clause in the block configuration makes potentially directly visible, then the given declaration is not directly visible in the block configuration or any of its configuration items. See 10.3 for more information.

For any name that is the label of a block statement appearing immediately within a given block, a corresponding block configuration may appear as a configuration item immediately within a block configuration corresponding to the given block. For any collection of names that are labels of instances of the same component appearing immediately within a given block, a corresponding component configuration may appear as a configuration item immediately within a block configuration corresponding to the given block.

For any name that is the label of a generate statement immediately within a given block, one or more corresponding block configurations may appear as configuration items immediately within a block configuration corresponding to the given block. Such block configurations apply to implicit blocks generated by that generate statement. If such a block configuration contains an index specification that is a discrete range, then the block configuration applies to those implicit block statements that are generated for the specified range of values of the corresponding generate parameter; the discrete range has no significance other than to define the set of generate statement parameter values implied by the discrete range. If such a block configuration contains an index specification that is a static expression, then the block configuration applies only to the implicit block statement generated for the specified value of the corresponding generate parameter. If no index specification appears in such a block configuration, then it applies to exactly one of the following sets of blocks:

-- All implicit blocks (if any) generated by the corresponding generate statement, if and only if the corresponding generate statement has a generation scheme including the reserved word **for**

-- The implicit block generated by the corresponding generate statement, if and only if the corresponding generate statement has a generation scheme including the reserved word **if** and if the condition in the generate scheme evaluates to TRUE

-- No implicit or explicit blocks, if and only if the corresponding generate statement has a generation scheme including the reserved word **if** and the condition in the generate scheme evaluates to FALSE

If the block specification of a block configuration contains a generate statement label, and if this label contains an index specification, then it is an error if the generate statement denoted by the label does not have a generation scheme including the reserved word **for**.

Within a given block configuration, whether implicit or explicit, an implicit block configuration is assumed to appear for any block statement that appears within the block corresponding to the given block configuration, if no explicit block configuration appears for that block statement. Similarly, an implicit component configuration is assumed to appear for each component instance that appears within the block corresponding to the given block configuration, if no explicit component configuration appears for that instance. Such implicit configuration items are assumed to appear following all explicit configuration items in the block configuration.

It is an error if, in a given block configuration, more than one configuration item is defined for the same block or component instance.

NOTES

1--As a result of the rules described in the preceding paragraphs and in Section 10, a simple name that is visible by selection at the end of the declarative part of a given block is also visible by selection within any configuration item contained in a corresponding block configuration. If such a name is directly visible at the end of the given block declarative part, it will likewise be directly visible in the corresponding configuration items,unless a use clause for a different declaration with the same simple name appears in the corresponding configuration declaration, and the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the name will be directly visible within the corresponding configuration items except at those places that fall within the scope of the additional use clause (at which places neither name will be directly visible).

2--If an implicit configuration item is assumed to appear within a block configuration, that implicit configuration item will never contain explicit configuration items.

3--If the block specification in a block configuration specifies a generate statement label, and if this label contains an index specification that is a discrete range, then the direction specified or implied by the discrete range has no significance other than to define, together with the bounds of the range, the set of generate statement parameter values denoted by the range. Thus, the following two block configurations are equivalent:

```
for Adders(31 downto 0) ··· end for;

for Adders(0 to 31) ··· end for;
```

4--A block configuration may appear immediately within a configuration declaration only if the entity declaration denoted by the entity name of the enclosing configuration declaration has associated architectures. Furthermore, the block specification of the block configuration must denote one of these architectures.

Examples:

```
--A block configuration for a design entity:
```

```
     for ShiftRegStruct                 --  An architecture name.
          --  Configuration items
          --  for blocks and components
          --  within ShiftRegStruct.
     end for ;

--A block configuration for a block statement:

     for B1                             --  A block label
          --  Configuration items
          --  for blocks and components
          --  within block B1.
     end for ;
```

1.3.2 Component configuration

A component configuration defines the configuration of one or more component instances in a corresponding block.

```
     component_configuration ::=
        for component_specification
            [ binding_indication ; ]
            [ block_configuration ]
        end for ;
```

The component specification (see 5.2) identifies the component instances to which this component configuration applies. A component configuration that appears immediately within a given block configuration applies to component instances that appear immediately within the corresponding block.

It is an error if two component configurations apply to the same component instance.

If the component configuration contains a binding indication (see 5.2.1), then the component configuration implies a configuration specification for the component instances to which it applies. This implicit configuration specification has the same component specification and binding indication as that of the component configuration.

If a given component instance is unbound in the corresponding block, then any explicit component configuration for that instance that does not contain an explicit binding indication will contain an implicit, default binding indication (see 5.2.2). Similarly, if a given component instance is unbound in the corresponding block, then any implicit component configuration for that instance will contain an implicit, default binding indication.

It is an error if a component configuration contains an explicit block configuration and the component configuration does not bind all identified component instances to the same design entity.

Within a given component configuration, whether implicit or explicit, an implicit block configuration is assumed for the design entity to which the corresponding component instance is bound, if no explicit block configuration appears and if the corresponding component instance is fully bound.

Examples:

```
--A component configuration with binding indication:
```

```
     for all: IOPort
          use entity StdCells.PadTriState4 (DataFlow)
               port map (Pout=>A, Pin=>B, IO=>Dir, Vdd=>Pwr, Gnd=>Gnd) ;
     end for ;

--A component configuration containing block configurations:

     for D1: DSP
         for DSP_STRUCTURE
              --  Binding specified in design entity or else defaults.
             for Filterer
                  --  Configuration items for filtering components.
             end for ;
             for Processor
                  --  Configuration items for processing components.
             end for ;
         end for ;
     end for ;
```

NOTE--Therequirement that all component instances corresponding to a block configuration be bound to the same design entity makes the following configuration illegal:

```
     architecture A of E is
          component C is end component C;
          for L1: C use entity E1(X);
          for L2: C use entity E2(X);
     begin
         L1: C;
         L2: C;
     end architecture A;

     configuration Illegal of Work.E is
         for A
             for all: C
                   for X     --  Does not apply to the same design entity in all
instances of C.
                     ...
                   end for; -- X
             end for; -- C
         end for; -- A
     end configuration Illegal ;
```

Section 0

Overview of this standard

This section describes the purpose and organization of this standard, the IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1993).

0.1 Intent and scope of this document

The intent of this standard is to define VHDL accurately. Its primary audiences are the implementor of tools supporting the language and the advanced user of the language. Other users are encouraged to use commercially available books, tutorials, and classes to learn the language in some detail prior to reading this manual. These resources generally focus on how to use the language, rather than how a VHDL-compliant tool is required to behave.

At the time of its publication, this document was the authoritative definition of VHDL. From time to time, it may become necessary to correct and/or clarify portions of this standard. Such corrections and clarifications may be published in separate documents. Such documents modify this standard at the time of their publication and remain in effect until superseded by subsequent documents or until the standard is officially revised.

0.2 Structure and terminology of this document

This manual is organized into sections, each of which focuses on some particular area of the language. Every fifth line of each section, not including section headings, footers, and the section title, is numbered in the left margin. Within each section, individual constructs or concepts are discussed in each clause.

Each clause describing a specific construct begins with an introductory paragraph. Next, the syntax of the construct is described using one or more grammatical "productions."

A set of paragraphs describing the meaning and restrictions of the construct in narrative form then follow. Unlike many other IEEE standards, which use the verb "shall" to indicate mandatory requirements of the standard and "may" to indicate optional features, the verb "is" is used uniformly throughout this document. In all cases, "is" is to be interpreted as having mandatory weight.

Additionally, the word "must" is used to indicate mandatory weight. This word is preferred over the more common "shall," as "must" denotes a different meaning to different readers of this standard.

a. To the developer of tools that process VHDL, "must" denotes a requirement that the standard imposes. The resulting implementation is required to enforce the requirement and to issue an error if the requirement is not met by some VHDL source text.

b. To the VHDL model developer, "must" denotes that the characteristics of VHDL are natural consequences of the language definition. The model developer is required to adhere to the constraint implied by the characteristic.

c. To the VHDL model user, "must" denotes that the characteristics of the models are natural consequences of the language definition. The model user can depend on the characteristics of the model implied by its VHDL source text.

Finally, each clause may end with examples, notes, and references to other pertinent clauses.

0.2.1 Syntactic description

The form of a VHDL description is described by means of context-free syntax, using a simple variant of backus naur form; in particular:

a. Lowercased words in roman font, some containing embedded underlines, are used to denote syntactic categories, for example:

```
formal_port_list
```

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

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

```
array
```

Reserved words must 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:

```
letter_or_digit ::= letter | digit

choices  ::=  choice { | choice }
```

In the first instance, an occurrence of "letter_or_digit" can be replaced by either "letter" or "digit." In the second case, "choices" can be replaced by a list of "choice," separated by vertical bars (see item f for the meaning of braces).

e. Square brackets enclose optional items on the right-hand side of a production; thus the two following productions are equivalent:

```
return_statement ::= return [ expression ] ;

return_statement ::= return ; | return expression ;
```

Note, however, that the initial and terminal square brackets in the right-hand side of the production for signatures (in 2.3.2) are part of the syntax of signatures and do not indicate that the entire right-hand side is optional.

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. Thus, the following two productions are equivalent:

```
term ::= factor { multiplying_operator factor }
```

```
term ::= factor | term multiplying_operator factor
```

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, *type*_name and *subtype*_name are both syntactically equivalent to name alone.

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

0.2.2 Semantic description

The meaning and restrictions of a particular construct are described with a set of narrative rules immediately following the syntactic productions. In these rules, an italicized term indicates the definition of that term and identifiers appearing entirely in uppercase refer to definitions in package STANDARD (see 14.2).

The following terms are used in these semantic descriptions with the following meaning:

```
erroneous          The condition described represents an ill-formed description;
implementations are however,
                   not required to detect and report this condition.

                   Conditions are deemed erroneous only when it is impossible in
general to detect the condition
                   During The Processing Of The Language.

error              The condition described represents an ill-formed description;
implementations are required to
                   detect the condition and report an error to the user of the tool.

illegal            A synonym for "error."

legal              The condition described represents a well-formed description.
```

0.2.3 Front matter, examples, notes, references, and annexes

Prior to this section are several pieces of introductory material; following the final section 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 VHDL.

Some clauses of this standard contain examples, notes, and cross-references to other clauses of the manual; these parts always appear at the end of a clause. Examples are meant to illustrate the possible forms of the construct described. Illegal examples are italicized. 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 this standard, notes are set as enumerated paragraphs in a font smaller than the rest of the text. Cross-references are meant to guide the user to other relevant clauses of the manual. Examples, notes, and cross-references are not part of the definition of the language.

Section 14

Predefined language environment

This section describes the predefined attributes of VHDL and the packages that all VHDL implementations must provide.

14.1 Predefined Attributes

Predefined attributes denote values, functions, types, and ranges associated with various kinds of named entities. These attributes are described below. For each attribute, the following information is provided:

-- The kind of attribute: value, type, range, function, or signal.

-- The prefixes for which the attribute is defined.

-- A description of the parameter or argument, if one exists.

-- The result of evaluating the attribute, and the result type (if applicable).

-- Any further restrictions or comments that apply.

T'BASE

Kind:	Type.
Prefix:	Any type or subtype T.
Result:	The base type of T.
Restrictions:	This attribute is allowed only as the prefix of the name of another attribute; for example, T'BASE'LEFT.

T'LEFT

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The left bound of T.

T'RIGHT

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The right bound of T.

T'HIGH

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The upper bound of T.

T'LOW

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The lower bound of T.

T'ASCENDING

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Type Boolean
Result:	TRUE if T is defined with an ascending range; FALSE otherwise.

T'IMAGE(X)

Kind:	Function.
Prefix:	Any scalar type or subtype T.
Parameter:	An expression whose type is the base type of T.

Result Type:	Type String.
Result:	The string representation of the parameter value, without leading or trailing whitespace. If T is an enumeration type or subtype and the parameter value is either an extended identifier or a character literal, the result is expressed with both a leading and trailing reverse solidus (backslash) (in the case of an extended identifier) or apostrophe (in the case of a character literal); in the case of an extended identifier that has a backslash, the backslash is doubled in the string representation. If T is an enumeration type or subtype and the parameter value is a basic identifier, then the result is expressed in lowercase characters. If T is a numeric type or subtype, the result is expressed as the decimal representation of the parameter value without underlines or leading or trailing zeros (except as necessary to form the image of a legal literal with the proper value); moreover, an exponent may (but is not required to) be present and the language does not define under what conditions it is or is not present. If the exponent is present, the "e" is expressed as a lowercase character. If T is a physical type or subtype, the result is expressed in terms of the primary unit of T unless the base type of T is TIME, in which case the result is expressed in terms of the resolution limit(see 3.1.3.1); in either case, if the unit is a basic identifier, the image of the unit is expressed in lowercase characters. If T is a floating point type or type, the number of digits to the right of the decimal point corresponds to the standard form generated when the DIGITS parameter to TextIO. Write for type REAL is set to 0 (see 14.3). The result never contains the replacement characters described in 13.10 .
Restrictions:	It is an error if the parameter value does not belong to the subtype implied by the prefix.

T'VALUE(X)

Kind:	Function.
Prefix:	Any scalar type or subtype T.
Parameter:	An expression of type String.
Result Type:	The base type of T.
Result:	The value of T whose string representation (as defined in Section 13) is given by the parameter. Leading and trailing whitespace is allowed and ignored. If T is a numeric type or subtype, the parameter may be expressed either as a decimal literal or as a based literal. If T is a physical type or subtype, the parameter may be expressed using a string representation of any of the unit names of T, with or without a leading abstract literal. The parameter must have whitespace between any abstract literal and the unit name. The replacement characters of 13.10 are allowed in the parameter.
Restrictions:	It is an error if the parameter is not a valid string representation of a literal of type T or if the result does not belong to the subtype implied by T.

T'POS(X)

	Kind:	Function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result Type:	*universal_integer*.
	Result:	The position number of the value of the parameter.
T'VAL(X)		
	Kind:	Function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression of any integer type.
	Result Type:	The base type of T.
	Result:	The value whose position number is the *universal_integer* value corresponding to X.
	Restrictions:	It is an error if the result does not belong to the range T'LOW **to** T'HIGH.
T'SUCC(X)		
	Kind:	Function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result Type:	The base type of T.
	Result:	The value whose position number is one greater than that of the parameter.
	Restrictions:	An error occurs if X equals T'HIGH or if X does not belong to the range T'LOW **to** T'HIGH.
T'PRED(X)		
	Kind:	Function.
	Prefix:	Any discrete or physical type or subtype T.
	Parameter:	An expression whose type is the base type of T.
	Result Type:	The base type of T.
	Result:	The value whose position number is one less than that of the parameter.
	Restrictions:	An error occurs if X equals T'LOW or if X does not belong to the range T'LOW **to** T'HIGH.
T'LEFTOF(X)		
	Kind:	Function.
	Prefix:	Any discrete or physical type or subtype T.

Parameter:	An expression whose type is the base type of T.
Result Type:	The base type of T.
Result:	The value that is to the left of the parameter in the range of T.
Restrictions:	An error occurs if X equals T'LEFT or if X does not belong to the range T'LOW **to** T'HIGH.

T'RIGHTOF(X)

Kind:	Function.
Prefix:	Any discrete or physical type or subtype T.
Parameter:	An expression whose type is the base type of T.
Result Type:	The base type of T.
Result:	The value that is to the right of the parameter in the range of T.
Restrictions:	An error occurs if X equals T'RIGHT or if X does not belong to the range T'LOW **to** T'HIGH.

A'LEFT [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the left bound of the Nth index range of A.
Result:	Left bound of the Nth index range of A. (If A is an alias for an array object, then the result is the left bound of the Nth index range from the declaration of A, not that of the object.)

A'RIGHT [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the Nth index range of A.
Result:	Right bound of the Nth index range of A. (If A is an alias for an array object, then the result is the right bound of the Nth index range from the declaration of A, not that of the object.)

A'HIGH [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the Nth index range of A.
Result:	Upper bound of the Nth index range of A. (If A is an alias for an array object, then the result is the upper bound of the Nth index range from the declaration of A, not that of the object.)

A'LOW [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the Nth index range of A.
Result:	Lower bound of the Nth index range of A. (If A is an alias for an array object, then the result is the lower bound of the Nth index range from the declaration of A, not that of the object.)

A'RANGE [(N)]

Kind:	Range.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	The type of the Nth index range of A.
Result:	The range A'LEFT(N) **to** A'RIGHT(N) if the Nth index range of A is ascending, or the range A'LEFT(N) **downto** A'RIGHT(N) if the Nth index range of A is descending. (If A is an alias for an array object, then the result is determined by the Nth index range from the declaration of A, not that of the object.)

A'REVERSE_RANGE [(N)]

Kind:	Range.

	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
	Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result Type:	The type of the Nth index range of A.
	Result:	The range A'RIGHT(N) **downto** A'LEFT(N) if the Nth index range of A is ascending, or the range A'RIGHT(N) **to** A'LEFT(N) if the Nth index range of A is descending. (If A is an alias for an array object, then the result is determined by the Nth index range from the declaration of A, not that of the object.)
A'LENGTH [(N)]		
	Kind:	Value.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype
	Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result Type:	*universal_integer*.
	Result:	Number of values in the Nth index range; i.e., if the Nth index range of A is a null range, then the result is 0. Otherwise, the result is the value of T'POS(A'HIGH(N)) - T'POS(A'LOW(N)) + 1, where T is the subtype of the Nth index of A.
A'ASCENDING [(N)]		
	Kind:	Value.
	Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
	Parameter:	A locally static expression of type *universal integer*, the value of which must be greater than zero and must not exceed the dimensionality of A. If omitted, it defaults to 1.
	Result Type:	Type Boolean.
	Result:	TRUE if the Nth index range of A is defined with an ascending range; FALSE otherwise.
S'DELAYED [(T)]		
	Kind:	Signal.
	Prefix:	Any signal denoted by the static signal name S.
	Parameter:	A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.

Result Type: The base type of S.

Result: A signal equivalent to signal S delayed T units of time. The value of S'DELAYED(t) at time T_n is always equal to the value of S at time T_{n-t}. Specifically:

Let R be of the same subtype as S, let T >= 0 ns, and let P be a process statement of the form

```
P:      process (S)
        begin
           R <= transport S after T;
        end process ;
```

Assuming that the initial value of R is the same as the initial value of S, then the attribute 'DELAYED is defined such that S'DELAYED(T) = R for any T.

S'STABLE [(T)]

Kind: Signal.

Prefix: Any signal denoted by the static signal name S.

Parameter: A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.

Result Type: Type Boolean.

Result: A signal that has the value TRUE when an event has not occurred on signal S for T units of time, and the value FALSE otherwise. (See 12.6.2 .)

S'QUIET [(T)]

Kind: Signal.

Prefix: Any signal denoted by the static signal name S

Parameter: A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.

Result Type: Type Boolean.

Result: A signal that has the value TRUE when the signal has been quiet for T units of time, and the value FALSE otherwise. (See 12.6.2 .)

S'TRANSACTION

Kind: Signal.

Prefix: Any signal denoted by the static signal name S.

Result Type: Type Bit.

Result: A signal whose value toggles to the inverse of its previous value in each simulation cycle in which signal S becomes active.

Restriction: A description is erroneous if it depends on the initial value of S'Transaction.

S'EVENT

Kind: Function.

Prefix: Any signal denoted by the static signal name S.

Result Type: Type Boolean.

Result: A value that indicates whether an event has just occurred on signal S. Specifically:

For a scalar signal S, S'EVENT returns the value TRUE if an event has occurred on S during the current simulation cycle; otherwise, it returns the value FALSE.

For a composite signal S, S'EVENT returns TRUE if an event has occurred on any scalar subelement of S during the current simulation cycle; otherwise, it returns FALSE.

S'ACTIVE

Kind: Function.

Prefix: Any signal denoted by the static signal name S.

Result Type: Type Boolean.

Result: A value that indicates whether signal S is active. Specifically:

For a scalar signal S, S'ACTIVE returns the value TRUE if signal S is active during the current simulation cycle; otherwise, it returns the value FALSE.

For a composite signal S, S'ACTIVE returns TRUE if any scalar subelement of S is active during the current simulation cycle; otherwise, it returns FALSE.

S'LAST_EVENT

Kind: Function.

Prefix: Any signal denoted by the static signal name S.

Result Type: Type Time.

Result: The amount of time that has elapsed since the last event occurred on signal S. Specifically:

For a signal S, S'LAST_EVENT returns the smallest value T of type TIME such that S'EVENT = True during any simulation cycle at time NOW - T, if such value exists; otherwise, it returns TIME'HIGH.

S'LAST_ACTIVE

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Time.
Result:	The amount of time that has elapsed since the last time at which signal S was active. Specifically:

For a signal S, S'LAST_ACTIVE returns the smallest value T of type TIME such that S'ACTIVE = True during any simulation cycle at time NOW - T, if such value exists; otherwise, it returns TIME'HIGH.

S'LAST_VALUE

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	The base type of S.
Result:	The previous value of S, immediately before the last change of S.

S'DRIVING

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Boolean.
Result:	If the prefix denotes a scalar signal, the result is False if the current value of the driver for S in the current process is determined by the null transaction; True otherwise. If the prefix denotes a composite signal,the result is True if and only if R'DRIVING is True for every scalar subelement R of S; False otherwise. If the prefix denotes a null slice of a signal, the result is True.
Restrictions:	This attribute is available only from within a process, a concurrent statement with an equivalent process, or a subprogram. If the prefix denotes a port, it is an error if the port does not have a mode of **inout**, **out**, or **buffer**. It is also an error if the attribute name appears in a subprogram body that is not a declarative item contained within a process statement and the prefix is not a formal parameter of the given subprogram or of a parent of that subprogram. Finally, it is an error if the prefix denotes a subprogram formal parameter whose mode is not **inout** or **out**.

S'DRIVING_VALUE

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	The base type of S.

Result:	If S is a scalar signal S, the result is the current value of the driver for S in the current process. If S is a composite signal, the result is the aggregate of the values of R'DRIVING_VALUE for each element R of S. If Sis a null slice, the result is a null slice.
Restrictions:	This attribute is available only from within a process, a concurrent statement with an equivalent process, or a subprogram. If the prefix denotes a port, it is an error if the port does not have a mode of **inout**, **out**, or **buffer**. It is also an error if the attribute appears in a subprogram body that is not a declarative item contained within a process statement and the prefix is not a formal parameter of the given subprogram or of a parent of that subprogram. Finally, it is an error if the prefix denotes a subprogram formal parameter whose mode is not **inout** or **out**, or if S'DRIVING is False at the time of the evaluation of S'DRIVING_VALUE.

E'SIMPLE_NAME

Kind:	Value.
Prefix:	Any named entity as defined in 5.1 .
Result Type:	Type String.
Result:	The simple name, character literal, or operator symbol of the named entity, without leading or trailing whitespace or quotation marks but with apostrophes (in the case of a character literal) and both a leading and trailing reverse solidus (backslash) (in the case of an extended identifier).In the case of a simple name or operator symbol, the characters are converted to their lowercase equivalents. In the case of an extended identifier, the case of the identifier preserved, and any reverse solidus characters appearing as part of the identifier are represented with two consecutive reverse solidus characters.

E'INSTANCE_NAME

Kind:	Value.
Prefix:	Any named entity other than the local ports and generics of a component declaration.
Result Type:	Type String.

The result string has the following syntax:

```
instance_name ::=  package_based_path | full_instance_based_path

    package_based_path ::=
        leader library_logical_name leader package_simple_name  leader
            [ local_item_name ]

    full_instance_based_path ::=  leader full_path_to_instance [ local_item_name ]

    full_path_to_instance ::=  { full_path_instance_element leader }

    local_item_name ::=
        simple_name
        character_literal
        operator_symbol

    full_path_instance_element ::=
        [ component_instantiation_label @ ]
            entity_simple_name ( architecture_simple_name )
        | block_label
        | generate_label
        | process_label
        | loop_label
        | subprogram_simple_name

    generate_label ::=  generate_label [ ( literal ) ]

    process_label ::=  [ process_label ]

    leader ::=  :
```

Package-based paths identify items declared within packages. Full-instance-based paths identify items within an elaborated design hierarchy.

A library logical name denotes a library; see 11.2 . Since multiple logical names may denote the same library, the library logical name may not be unique.

There is one full path instance element for each component instantiation, block statement, generate statement, process statement, or subprogram

body in the design hierarchy between the root design entity and the named entity denoted by the prefix.

In a full path instance element, the architecture simple name must denote an architecture associated with the entity interface designated by the entity simple name; furthermore, the component instantiation label (and the commercial at following it) are required unless the entity simple name and the architecture simple name together denote the root design entity.

The literal in a generate label is required if the label denotes a generate statement with a for generation scheme; the literal must denote one of the values of the generate parameter.

A process statement with no label is denoted by an empty process label.

All characters in basic identifiers appearing in the result are converted to their lowercase equivalents. Both a leading and trailing reverse solidus surround an extended identifier appearing in the result; any reverse solidus characters appearing as part of the identifier are represented with two consecutive reverse solidus characters.

E'PATH_NAME

Kind:	Value.
Prefix:	Any named entity other than the local ports and generics of a component declaration.
Result Type:	Type String.
Result:	A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, excluding the name of instantiated design entities. Specifically:

The result string has the following syntax:

```
path_name ::=  package_based_path | instance_based_path

instance_based_path ::=
    leader path_to_instance [ local_item_name ]

 path_to_instance ::=  { path_instance_element leader }

 path_instance_element  ::=
    component_instantiation_label
    | entity_simple_name
    | block_label
    | generate_label
    | process_label
    | subprogram_simple_name
```

Package-based paths identify items declared within packages. Full- instance - based paths identify items within an elaborated design hierarchy.

There is one path instance element for each component instantiation, block statement, generate statement, process statement, or subprogram body in the design hierarchy between the top design entity and the named entity denoted by the prefix.

Examples:

```
     library Lib:                                  --  All design units are in this
library:
     package P is                                  --  P'PATH_NAME = ":lib:p:"
                                                       --P'INSTANCE_NAME = ":lib:p:"

         procedure Proc (F: inout INTEGER);        --  Proc'PATH_NAME = ":lib:p:proc"
                                                            --Proc'INSTANCE_NAME =
":lib:p:proc"

         constant C: INTEGER := 42;                --  C'PATH_NAME = ":lib:p:c"
     end package P;                                --  C'INSTANCE_NAME = ":lib:p:c"

     package body P is
```

```
        procedure Proc (F: inout INTEGER) is
            variable x: INTEGER;                        --  x'PATH_NAME =
":lib:p:proc:x"
        begin                                           --  x'INSTANCE_NAME =
":lib:p:proc:x"
            .
            .
            .
        end;
    end;

    library Lib;
    use Lib.P.all;                                       --  Assume that E is in Lib
and
    entity E is                                          --  E is the top-level design
entity:
                                                         --  E'PATH_NAME = ":e:"
                                                         --  E'INSTANCE_NAME =
":e(a):"
        generic (G: INTEGER);                            --  G'PATH_NAME = ":e:g"
                                                         --  G'INSTANCE_NAME =
":e(a):g"
        port (P: in INTEGER);                            --  P'PATH_NAME = ":e:p"
    end entity E;                                        --  P'INSTANCE_NAME =
":e(a):p"

    architecture A of E is
        signal S: BIT_VECTOR (1 to G);                    --  S'PATH_NAME = ":e:s"
                                                          --  S'INSTANCE_NAME =
":e(a):s"
        procedure Proc1 (signal sp1: NATURAL; C: out INTEGER) is
                                                          --  Proc1'PATH_NAME =
":e:proc1:"
                                                          --  Proc1'INSTANCE_NAME
=:e(a):proc1:"
                                                          --  C'PATH_NAME =
":e:proc1:c"
```

```
                                                        --  C'INSTANCE_NAME =
":e(a):proc1:c"
        variable max: DELAY_LENGTH;                     --  max'PATH_NAME =
":e:proc1:max"
                                                        --  max'INSTANCE_NAME =
                                                        --  ":e(a):proc1:max"

        begin
            max := sp1 * ns;
            wait on sp1 for max;
            c := sp1;
        end procedure Proc1;

    begin
    p1: process
            variable T: INTEGER := 12;                  --  T'PATH_NAME =  ":e:p1:t"
        begin                                           --  T'INSTANCE_NAME =
":e(a):p1:t"
            .
            .
            .
        end process p1;

        process
            variable T: INTEGER := 12;                  --  T'PATH_NAME =  ":e::t"
        begin                                           --  T'INSTANCE_NAME =
":e(a)::t"
            .
            .
            .
        end process ;
    end architecture;

    entity Bottom is
        generic (GBottom : INTEGER);
        port     (PBottom : INTEGER);
    end entity Bottom;
```

```
    architecture BottomArch of Bottom is
        signal SBottom : INTEGER;
    begin
        ProcessBottom : process
            variable V : INTEGER;
        begin
            if GBottom = 4 then
                assert V'Simple_Name = "v"
                    and V'Path_Name = ":top:b1:b2:g1(4):b3:l1:processbottom:v"
                    and V'Instance_Name =
":top(top):b1:b2:g1(4):b3:l1@bottom(bottomarch):processbottom:v";
                assert GBottom'Simple_Name = "bottom"
                    and GBottom'Path_Name = ":top:b1:b2:g1(4):b3:l1:gbottom"
                    and GBottom'Instance_Name =
":top(top):b1:b2:g1(4):b3:l1@bottom(bottomarch):gbottom";
            elsif GBottom = -1 then
                assert V'Simple_Name = "v"
                    and V'Path_Name = ":top:l2:processbottom:v"
                    and V'Instance_Name =
":top(top):l2@bottom(bottomarch):processbottom:v";
                assert GBottom'Simple_Name = "gbottom"
                    and GBottom'Path_Name = "top:l2:gbottom"
                    and GBottom'Instance_Name =
                     ":top(top):l2@bottom(bottomarch):gbottom";
            end if;
        wait;
        end process ProcessBottom;
    end architecture BottomArch;

    entity Top is end Top;

    architecture Top of Top is
        component BComp is
            generic (GComp : INTEGER)
            port      (PComp : INTEGER);
        end component BComp;
```

```
        signal S : INTEGER;
    begin
        B1 : block
            signal S : INTEGER;

        begin
            B2 : block
                signal S : INTEGER;
            begin
                G1 : for I in 1 to 10 generate
                    B3 : block
                         signal S : INTEGER;
                         for L1 : BComp use entity Work.Bottom(BottomArch)
                              generic map    (GBottom => GComp)
                              port map       (PBottom => PComp);

                    begin
                         L1 : BComp generic map (I) port map (S);
                         P1 : process
                              variable V : INTEGER;
                         begin
                              if I = 7 then
                                   assert V'Simple_Name = "v"
                                       and V'Path_Name = ":top:b1:b2:g1(7):b3:p1:v"
                                       and V'Instance_Name =
":top(top):b1:b2:g1(7):b3:p1:v";
                                   assert P1'Simple_Name = "p1"
                                       and P1'Path_Name = ":top:b1:b2:g1(7):b3:p1:"
                                       and P1'Instance_Name =
":top(top):b1:b2:g1(7):b3:p1:";
                                   assert S'Simple_Name = "s"
                                       and S'Path_Name = ":top:b1:b2:g1(7):b3:s"
                                       and S'Instance_Name =
":top(top):b1:b2:g1(7):b3:s";
                                   assert B1.S'Simple_Name = "s"
                                       and B1.S'Path_Name = ":top:b1:s"
```

```
                                        and B1.S'Instance_Name = ":top(top):b1:s";
                            end if;
                            wait;
                        end process P1;
                    end block B3;
                end generate;
            end block B2;
        end block B1;
        L2 : BComp generic map (-1) port map (S);
    end architecture Top;

    configuration TopConf of Top is
        for Top
            for L2 : BComp use
                entity Work.Bottom(BottomArch)
                    generic map   (GBottom => GComp)
                    port map      (PBottom => PComp);
            end for;
        end for;
    end configuration TopConf;
```

NOTES

1--The relationship between the values of the LEFT, RIGHT, LOW, and HIGH attributes is expressed in the following table:

		Ascending range	Descending range
T'LEFT	=	T'LOW	T'HIGH
T'RIGHT	=	T'HIGH	T'LOW

2--Since the attributes S'EVENT, S'ACTIVE, S'LAST_EVENT, S'LAST_ACTIVE, and S'LAST_VALUE are functions, not signals, they cannot cause the execution of a process, even though the value returned by such a function may change dynamically. It is thus recommended that the equivalent signal-valued attributes S'STABLE and S'QUIET, or expressions involving those attributes, be used in concurrent contexts such as guard expressions or concurrent signal assignments. Similarly, function STANDARD.NOW should not be used in concurrent contexts.

3--S'DELAYED(0 ns) is not equal to S during any simulation cycle where S'EVENT is true.

4--S'STABLE(0 ns) = (S'DELAYED(0 ns) = S), and S'STABLE(0 ns) is FALSE only during a simulation cycle in which S has had a transaction.

5--For a given simulation cycle, S'QUIET(0 ns) is TRUE if and only if S is quiet for that simulation cycle.

6--If S'STABLE(T) is FALSE, then, by definition, for some t where 0 ns <= t<= T, S'DELAYED(t) /= S.

7--If T_s is the smallest value such that S'STABLE (T_s) is FALSE, then for all t where 0 ns <= t < Ts, S'DELAYED(t) = S.

8--S'EVENT should not be used within a postponed process (or a concurrent statement that has an equivalent postponed process) to determine if the prefix signal S caused the process to resume. However, S'LAST_EVENT = 0 ns can be used for this purpose.

9--The values of E'PATH_NAME and E'INSTANCE_NAME are not unique. Specifically, named entities in two different, unlabelled processes may have the same path names or instance names. Overloaded subprograms, and named entities within them, may also have the same path names or instance names.

10--If the prefix to the attributes 'SIMPLE_NAME, 'PATH_NAME, or 'INSTANCE_NAME denotes an alias, the result is respectively the simple name, path name or instance name of the alias. See 6.6 .

11--For all values V of any scalar type T except a real type, the following relation holds:

```
V = T'Value(T'Image(V))
```

14.2 Package STANDARD

Package STANDARD predefines a number of types, subtypes, and functions. An implicit context clause naming this package is assumed to exist at the beginning of each design unit. Package STANDARD may not be modified by the user.

The operators that are predefined for the types declared for package STANDARD are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *universal_integer*), formal parameters, and undefined information (such as *implementation_defined*).

```
package STANDARD is
```

```
 --  Predefined enumeration types:

 type BOOLEAN is (FALSE, TRUE);

 --  The predefined operators for this type are as follows:

 --  function "and"     (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "or"      (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "nand"    (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "nor"     (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "xor"     (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "xnor"    (anonymous, anonymous: BOOLEAN)return BOOLEAN;

 --  function "not"     (anonymous: BOOLEAN) return BOOLEAN;

 --  function "="       (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "/="      (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "<"       (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function "<="      (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function ">"       (anonymous, anonymous: BOOLEAN)return BOOLEAN;
 --  function ">="      (anonymous, anonymous: BOOLEAN)return BOOLEAN;


type BIT is ('0', '1');

 --  The predefined operators for this type are as follows:

 --  function "and"     (anonymous, anonymous: BIT) return BIT;
 --  function "or"      (anonymous, anonymous: BIT) return BIT;
 --  function "nand"    (anonymous, anonymous: BIT) return BIT;
 --  function "nor"     (anonymous, anonymous: BIT) return BIT;
 --  function "xor"     (anonymous, anonymous: BIT) return BIT;
 --  function "xnor"    (anonymous, anonymous: BIT) return BIT;

 --  function "not"     (anonymous: BIT) return BIT;

 --  function "="       (anonymous, anonymous: BIT) return BOOLEAN;
```

```
--  function "/="      (anonymous, anonymous: BIT) return BOOLEAN;
--  function "<"       (anonymous, anonymous: BIT) return BOOLEAN;
--  function "<="      (anonymous, anonymous: BIT) return BOOLEAN;
--  function ">"       (anonymous, anonymous: BIT) return BOOLEAN;
--  function ">="      (anonymous, anonymous: BIT) return BOOLEAN;
```

```
type CHARACTER is(
```

NUL,	SOH,	STX,	ETX,	EOT,	ENQ,	ACK,	BEL,
BS,	HT,	LF,	VT,	FF,	CR,	SO,	SI,
DLE,	DC1,	DC2,	DC3,	DC4,	NAK,	SYN,	ETB,
CAN,	EM,	SUB,	ESC,	FSP,	GSP,	RSP,	USP,
' ',	'!',	'"',	'#',	'$',	'%',	'&',	''',
'(',	')',	'*',	'+',	',',	'-',	'.',	'/',
'0',	'1',	'2',	'3',	'4',	'5',	'6',	'7',
'8',	'9',	':',	';',	'<',	'=',	'>',	'?',
'@',	'A',	'B',	'C',	'D',	'E',	'F',	'G',
'H',	'I',	'J',	'K',	'L',	'M',	'N',	'O',
'P',	'Q',	'R',	'S',	'T',	'U',	'V',	'W',
'X',	'Y',	'Z',	'[',	'\',	']',	'^',	'_',
'`',	'a',	'b',	'c',	'd',	'e',	'f',	'g',
'h',	'i',	'j',	'k',	'l',	'm',	'n',	'o',
'p',	'q',	'r',	's',	't',	'u',	'v',	'w',
'x',	'y',	'z',	'{',	'|',	'}',	'~',	DEL,
C128,	C129,	C130,	C131,	C132,	C133,	C134,	C135,
C136,	C137,	C138,	C139,	C140,	C141,	C142,	C143,
C144,	C145,	C146,	C147,	C148,	C149,	C150,	C151,
C152,	C153,	C154,	C155,	C156,	C157,	C158,	C159,
' ',*	'¡',	'¢',	'£',	'¤',	'¥',	'¦',	'§',
'¨',	'©',	'ª',	'«',	'¬',	'-',[+]	'®',	'¯',

```
                'º',        '±',        '²',        '³',        '"',        'µ',        '¶',        '·',
                '¸',        '¹',        'º',        '»',        '¼',        '½',        '¾',        '¿',
                'À',        'Á',        'Â',        'Ã',        'Ä',        'Å',        'Æ',        'Ç',
                'È',        'É',        'Ê',        'Ë',        'Ì',        'Í',        'Î',        'Ï',
                'Ð',        'Ñ',        'Ò',        'Ó',        'Ô',        'Õ',        'Ö',        '×',
                'Ø',        'Ù',        'Ú',        'Û',        'Ü',        'Ý',        'Þ',        'ß',
                'à',        'á',        'â',        'ã',        'ä',        'å',        'æ',        'ç',
                'è',        'é',        'ê',        'ë',        'ì',        'í',        'î',        'ï',
                'ð',        'ñ',        'ò',        'ó',        'ô',        'õ',        'ö',        '÷',
                'ø',        'ù',        'ú',        'û',        'ü',        'ý',        'þ',        'ÿ');

--  The predefined operators for this type are as follows:

--  function "="      (anonymous, anonymous: CHARACTER) return BOOLEAN;
--  function "/="     (anonymous, anonymous: CHARACTER) return BOOLEAN;
--  function "<"      (anonymous, anonymous: CHARACTER) return BOOLEAN;
--  function "<="     (anonymous, anonymous: CHARACTER) return BOOLEAN;
--  function ">"      (anonymous, anonymous: CHARACTER) return BOOLEAN;
--  function ">="     (anonymous, anonymous: CHARACTER) return BOOLEAN;

type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);

--  The predefined operators for this type are as follows:

--  function "="      (anonymous, anonymous: SEVERITY_LEVEL)return BOOLEAN;
--  function "/="     (anonymous, anonymous: SEVERITY_LEVEL)return BOOLEAN;
--  function "<"      (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
--  function "<="     (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
--  function ">"      (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
--  function ">="     (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;

--  type universal_integer  is range implementation_defined;

--  The predefined operators for this type are as follows:
```

```
        --  function "="     (anonymous, anonymous: universal_integer) return
BOOLEAN;
        --  function "/="    (anonymous, anonymous: universal_integer) return
BOOLEAN;
        --  function "<"     (anonymous, anonymous: universal_integer) return
BOOLEAN;
        --  function "<="    (anonymous, anonymous: universal_integer) return
BOOLEAN;
        --  function ">"     (anonymous, anonymous: universal_integer) return
BOOLEAN;
        --  function ">="    (anonymous, anonymous: universal_integer) return
BOOLEAN;
        --  function "+"     (anonymous: universal_integer) return
universal_integer;
        --  function "-"     (anonymous: universal_integer) return
universal_integer;
        --  function "abs"   (anonymous: universal_integer) return
universal_integer;
        --  function "+"     (anonymous,  universal_integer)
        --                                   return universal_integer;
        --  function "-"     (anonymous,  universal_integer)
        --                                   return universal_integer;
        --  function "abs"     (anonymous,  universal_integer)
        --                                   return universal_integer;
        --  function "+"     (anonymous, anonymous: universal_integer)
        --                                   return universal_integer;
        --  function "-"     (anonymous, anonymous: universal_integer)
        --                                   return universal_integer;
        --  function "*"     (anonymous, anonymous: universal_integer)
        --                                   return universal_integer;

        --  function "/"     (anonymous,anonymous:  universal_integer)
        --                                   return universal_integer;
        --  function "mod"   (anonymous, anonymous: universal_integer)
        --                                   return universal_integer;
        --  function "rem"   (anonymous, anonymous: universal_integer)
```

```
        --                                      return universal_integer;

        --  type universal_real  is range implementation_defined;

        --  The predefined operators for this type are as follows:

        --  function "="      (anonymous, anonymous:universal_real) return BOOLEAN;
        --  function "/="     (anonymous, anonymous:universal_real) return BOOLEAN;
        --  function "<"      (anonymous, anonymous:universal_real) return BOOLEAN;
        --  function "<="     (anonymous, anonymous:universal_real) return BOOLEAN;
        --  function ">"      (anonymous, anonymous:universal_real) return BOOLEAN;
        --  function ">="     (anonymous, anonymous:universal_real) return BOOLEAN;

        --  function "+"      (anonymous: universal_real) return universal_real;
        --  function "-"      (anonymous: universal_real) return universal_real;
        --  function "abs"    (anonymous: universal_real) return universal_real;

        --  function "+"      (anonymous, anonymous:universal_real) return
universal_real;
        --  function "-"      (anonymous, anonymous:universal_real) return
universal_real;
        --  function "*"      (anonymous, anonymous:universal_real) return
universal_real;
        --  function "/"      (anonymous, anonymous:universal_real) return
universal_real;

        --  function "*"      (anonymous: universal_real;anonymous:
universal_integer)
        --                                      return universal_real;
        --  function "*"      (anonymous: universal_integer;anonymous:
universal_real)
        --                                      return universal_real;
        --  function "/"      (anonymous: universal_real;anonymous:
universal_integer)
        --                                      return universal_real;
```

```
--  Predefined numeric types:

type INTEGER is range implementation_defined;

--  The predefined operators for this type are as follows:

--  function "**"     (anonymous: universal_integer; anonymous: INTEGER)
--                                    return universal_integer;
--  function "**"     (anonymous: universal_real; anonymous: INTEGER)
--                                    return universal_real;

--  function "="      (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function "/="     (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function "<"      (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function "<="     (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function ">"      (anonymous, anonymous: INTEGER) return BOOLEAN;
--  function ">="     (anonymous, anonymous: INTEGER) return BOOLEAN;

--  function "+"      (anonymous: INTEGER) return INTEGER;
--  function "-"      (anonymous: INTEGER) return INTEGER;
--  function "abs"    (anonymous: INTEGER) return INTEGER;

--  function "+"      (anonymous, anonymous: INTEGER) return INTEGER;
--  function "-"      (anonymous, anonymous: INTEGER) return INTEGER;
--  function "*"      (anonymous, anonymous: INTEGER) return INTEGER;
--  function "/"      (anonymous, anonymous: INTEGER) return INTEGER;
--  function "mod"    (anonymous, anonymous: INTEGER) return INTEGER;
--  function "rem"    (anonymous, anonymous: INTEGER) return INTEGER;

--  function "**"     (anonymous: INTEGER; anonymous: INTEGER)
--                                    return INTEGER;



type REAL is range implementation_defined;

--  The predefined operators for this type are as follows:
```

```
--  function "="      (anonymous, anonymous: REAL) return BOOLEAN;
--  function "/="     (anonymous, anonymous: REAL) return BOOLEAN;
--  function "<"      (anonymous, anonymous: REAL) return BOOLEAN;
--  function "<="     (anonymous, anonymous: REAL) return BOOLEAN;
--  function ">"      (anonymous, anonymous: REAL) return BOOLEAN;
--  function ">="     (anonymous, anonymous: REAL) return BOOLEAN;

--  function "+"      (anonymous: REAL) return REAL;
--  function "-       (anonymous: REAL) return REAL;
--  function "abs"    (anonymous: REAL) return REAL;

--  function "+"      (anonymous, anonymous: REAL) return REAL;
--  function "-"      (anonymous, anonymous: REAL) return REAL;
--  function "*"      (anonymous, anonymous: REAL) return REAL;
--  function "/"      (anonymous, anonymous: REAL) return REAL;

--  function "**"     (anonymous: REAL; anonymous: INTEGER) return REAL;


--  Predefined type TIME:

type TIME is range implementation_defined
    units

      fs;                       --  femtosecond
      ps     =   1000 fs;       --  picosecond
      ns     =   1000 ps;       --  nanosecond
      us     =   1000 ns;       --  microsecond
      ms     =   1000 us;       --  millisecond
      sec    =   1000 ms;       --  second
      min    =   60 sec;        --  minute
      hr     =   60 min;        --  hour
    end units;

--  The predefined operators for this type are as follows:

--  function "="      (anonymous, anonymous: TIME) return BOOLEAN;
```

```
        --  function "/="     (anonymous, anonymous: TIME) return BOOLEAN;
        --  function "<"      (anonymous, anonymous: TIME) return BOOLEAN;
        --  function "<="     (anonymous, anonymous: TIME) return BOOLEAN;
        --  function ">"      (anonymous, anonymous: TIME) return BOOLEAN;
        --  function ">="     (anonymous, anonymous: TIME) return BOOLEAN;

        --  function "+"      (anonymous: TIME) return TIME;
        --  function "-"      (anonymous: TIME) return TIME;
        --  function "abs"    (anonymous: TIME) return TIME;

        --  function "+"      (anonymous, anonymous: TIME) return TIME;
        --  function "-"      (anonymous, anonymous: TIME) return TIME;

        --  function "*"      (anonymous: TIME;     anonymous: INTEGER)    return
TIME;
        --  function "*"      (anonymous: TIME;     anonymous: REAL)       return
TIME;
        --  function "*"      (anonymous: INTEGER;  anonymous: TIME)       return
TIME;
        --  function "*"      (anonymous: REAL;     anonymous: TIME)       return
TIME;
        --  function "/"      (anonymous: TIME;     anonymous: INTEGER)    return
TIME;
        --  function "/"      (anonymous: TIME;     anonymous: REAL) return TIME;

        --  function "/"      (anonymous, anonymous: TIME) return universal_integer;

        subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;

        --  A function that returns the current simulation time, T_c (see   12.6.4
):
        impure function NOW return DELAY_LENGTH;

        --  Predefined numeric subtypes:
```

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;

subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;


--  Predefined array types:

type STRING is array (POSITIVE range <>)of CHARACTER;

--  The predefined operators for this type are as follows:

--  function "="      (anonymous, anonymous: STRING) return BOOLEAN;
--  function "/="     (anonymous, anonymous: STRING) return BOOLEAN;
--  function "<"      (anonymous, anonymous: STRING) return BOOLEAN;
--  function "<="     (anonymous, anonymous: STRING) return BOOLEAN;
--  function ">"      (anonymous, anonymous: STRING) return BOOLEAN;
--  function ">="     (anonymous, anonymous: STRING) return BOOLEAN;

--  function "&"      (anonymous: STRING;          anonymous: STRING)
--                                               return STRING;
--  function "&"      (anonymous: STRING;          anonymous: CHARACTER)
--                                               return STRING;
--  function "&"      (anonymous: CHARACTER;     anonymous: STRING)
--                                               return STRING;
--  function "&"      (anonymous: CHARACTER;     anonymous: CHARACTER)
--                                               return STRING;

type BIT_VECTOR is array (NATURAL range <>)of BIT;

--  The predefined operators for this type are as follows:

--  function "and"    (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
--  function "or"     (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
--  function "nand"   (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
--  function "nor"    (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
--  function "xor"    (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
--  function "xnor"   (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
```

```
--  function "not"    (anonymous: BIT_VECTOR) return BIT_VECTOR;

--  function "sll"    (anonymous: BIT_VECTOR; anonymous:INTEGER)
--                    return BIT_VECTOR;
--  function "srl"    (anonymous: BIT_VECTOR; anonymous:INTEGER)
--                    return BIT_VECTOR;
--  function "sla"    (anonymous: BIT_VECTOR; anonymous:INTEGER)
--                    return BIT_VECTOR;
--  function "sra"    (anonymous: BIT_VECTOR; anonymous:INTEGER)
--                    return BIT_VECTOR;
--  function "rol"    (anonymous: BIT_VECTOR; anonymous:INTEGER)
--                    return BIT_VECTOR;
--  function "ror"    (anonymous: BIT_VECTOR; anonymous:INTEGER)
--                    return BIT_VECTOR;

--  function "="      (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
--  function "/="     (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
--  function "<"      (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
--  function "<="     (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
--  function ">"      (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
--  function ">="     (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;

--  function "&"      (anonymous: BIT_VECTOR;     anonymous: BIT_VECTOR)
--                                              return BIT_VECTOR;
--  function "&"      (anonymous: BIT_VECTOR;     anonymous: BIT)
--                                              return BIT_VECTOR;
--  function "&"      (anonymous: BIT;            anonymous: BIT_VECTOR)
--                                              return BIT_VECTOR;
--  function "&"      (anonymous: BIT;            anonymous: BIT)
--                                              return BIT_VECTOR;


--  The predefined types for opening files:

type FILE_OPEN_KIND is (
   READ_MODE,                 --  Resulting access mode is read-only.
   WRITE_MODE,                --  Resulting access mode is write-only.
```

```
           APPEND_MODE);              --  Resulting access mode is write-only;
information
                                      --  is appended to the end of the existing
file.

        --  The predefined operators for this type are as follows:


        --  function "="      (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
        --  function "/="     (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
        --  function "<"      (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
        --  function "<="     (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
        --  function ">"      (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
        --  function ">="     (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;

        type FILE_OPEN_STATUS is (
           OPEN_OK,                   --  File open was successful.
           STATUS_ERROR,              --  File object was already open.
           NAME_ERROR,                --  External file not found or inaccessible.
           MODE_ERROR);               --  Could not open file with requested access
mode.

        --  The predefined operators for this type are as follows:

        --  function "="       (anonymous, anonymous: FILE_OPEN_STATUS)
                              return BOOLEAN;
        --  function "/="      (anonymous, anonymous: FILE_OPEN_STATUS)
                              return BOOLEAN;
        --  function "<"       (anonymous, anonymous: FILE_OPEN_STATUS)
                              return BOOLEAN;-
        --  function "<="      (anonymous, anonymous: FILE_OPEN_STATUS)
                              return BOOLEAN;
        --  function ">"       (anonymous, anonymous: FILE_OPEN_STATUS)
                              return BOOLEAN;
        --  function ">="      (anonymous, anonymous: FILE_OPEN_STATUS)
                              return BOOLEAN;
```

```
        --  The 'FOREIGN attribute:

                attribute FOREIGN: STRING;
    end STANDARD;
```

The 'FOREIGN attribute may be associated only with architectures (see 1.2) or with subprograms. In the latter case, the attribute specification must appear in the declarative part in which the subprogram is declared (see 2.1).

NOTES

1--The ASCII mnemonics for file separator (FS), group separator (GS), record separator (RS), and unit separator (US) are represented by FSP, GSP, RSP, and USP, respectively, in type CHARACTER in order to avoid conflict with the units of type TIME.

2--The declarative parts and statement parts of design entities whose corresponding architectures are decorated with the 'FOREIGN attribute and subprograms that are likewise decorated are subject to special elaboration rules. See 12.3 and 12.4 .

14.3 Package TEXTIO

Package TEXTIO contains declarations of types and subprograms that support formatted I/O operations on text files.

```
    package TEXTIO is

        --  Type definitions for text I/O:

          type LINE is access STRING;         --  A LINE is a pointer to a STRING
value.

          type TEXT is file of STRING;        --  A file of variable-length ASCII
records.

          type SIDE is (RIGHT, LEFT);         --  For justifying output data within
fields.

          subtype WIDTH is NATURAL;           --  For specifying widths of output
```

```
fields.

        --  Standard text files:

          file INPUT:     TEXT open READ_MODE     is "STD_INPUT";

          file OUTPUT:    TEXT open WRITE_MODE    is "STD_OUTPUT";

        --  Input routines for standard types:

          procedure READLINE (file F: TEXT; L: out LINE);

          procedure READ (L: inout LINE;     VALUE: out BIT;             GOOD: out
BOOLEAN);
          procedure READ (L: inout LINE;     VALUE: out BIT);

          procedure READ (L: inout LINE;     VALUE: out BIT_VECTOR;      GOOD: out
BOOLEAN);
          procedure READ (L: inout LINE;     VALUE: out BIT_VECTOR);

          procedure READ (L: inout LINE;     VALUE: out BOOLEAN;         GOOD: out
BOOLEAN);
          procedure READ (L: inout LINE;     VALUE: out BOOLEAN);

          procedure READ (L: inout LINE;     VALUE: out CHARACTER;       GOOD: out
BOOLEAN);
          procedure READ (L: inout LINE;     VALUE: out CHARACTER);

          procedure READ (L: inout LINE;     VALUE: out INTEGER;         GOOD: out
BOOLEAN);
          procedure READ (L: inout LINE;     VALUE: out INTEGER);

          procedure READ (L: inout LINE;     VALUE: out REAL;            GOOD: out
BOOLEAN);
          procedure READ (L: inout LINE;     VALUE: out REAL);

          procedure READ (L: inout LINE;     VALUE: out STRING;          GOOD: out
BOOLEAN);
```

```
        procedure READ (L: inout LINE;     VALUE: out STRING);

        procedure READ (L: inout LINE;     VALUE: out TIME;             GOOD: out
BOOLEAN);
        procedure READ (L: inout LINE;     VALUE: out TIME);


        --  Output routines for standard types:

        procedure WRITELINE (file F: TEXT; L: inout LINE);

        procedure WRITE (L: inout LINE;     VALUE: in BIT;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
        procedure WRITE (L: inout LINE;     VALUE: in BIT_VECTOR;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

        procedure WRITE (L: inout LINE;     VALUE: in BOOLEAN;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

        procedure WRITE (L: inout LINE;     VALUE: in CHARACTER;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

        procedure WRITE (L: inout LINE;     VALUE: in INTEGER;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

        procedure WRITE (L: inout LINE;     VALUE: in REAL;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                         DIGITS: in NATURAL:= 0);

        procedure WRITE (L: inout LINE;     VALUE: in STRING;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

        procedure WRITE (L: inout LINE;     VALUE: in TIME;
                         JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                         UNIT: in TIME:= ns);


        --  File position predicate:
```

```
    --function ENDFILE (file F: TEXT) return BOOLEAN;

    end TEXTIO;
```

Procedures READLINE and WRITELINE declared in package TEXTIO read and write entire lines of a file of type TEXT. Procedure READLINE causes the next line to be read from the file and returns as the value of parameter L an access value that designates an object representing that line. If parameter L contains a nonnull access value at the start of the call, the object designated by that value is deallocated before the new object is created. The representation of the line does not contain the representation of the end of the line. It is an error if the file specified in a call to READLINE is not open or, if open, the file has an access mode other than read-only (see 3.4.1). Procedure WRITELINE causes the current line designated by parameter L to be written to the file and returns with the value of parameter L designating a null string. If parameter L contains a null access value at the start of the call, then a null string is written to the file. It is an error if the file specified in a call to WRITELINE is not open or, if open, the file has an access mode other than write-only.

The language does not define the representation of the end of a line. An implementation must allow all possible values of types CHARACTER and STRING to be written to a file. However, as an implementation is permitted to use certain values of types CHARACTER and STRING as line delimiters, it may not be possible to read these values from a TEXT file.

Each READ procedure declared in package TEXTIO extracts data from the beginning of the string value designated by parameter L and modifies the value so that it designates the remaining portion of the line on exit.

The READ procedures defined for a given type other than CHARACTER and STRING begin by skipping leading *whitespace characters*. A whitespace character is defined as a space, a nonbreaking space, or a horizontal tabulation character (SP, NBSP, or HT). For all READ procedures, characters are then removed from L and composed into a string representation of the value of the specified type. Character removal and string composition stops when a character is encountered that cannot be part of the value according to the lexical rules of 13.2 ; this character is not removed from L and is not added to the string representation of the value. The READ procedures for types INTEGER and REAL also accept a leading sign; additionally, there can be no space between the sign and the remainder of the literal. The READ procedures for types STRING and BIT_VECTOR also terminate acceptance when VALUE'LENGTH characters have been accepted. Again using the rules of 13.2 , the accepted characters are then interpreted as a string representation of the specified type. The READ does not succeed if the sequence of characters removed from L is not a valid string representation of a value of the specified type or, in the case of types STRING and BIT_VECTOR, if the sequence does not contain VALUE'LENGTH characters.

The definitions of the string representation of the value for each data type are as follows:

-- The representation of a BIT value is formed by a single character, either1 or 0. No leading or trailing quotation characters are present.

-- The representation of a BIT_VECTOR value is formed by a sequence of characters, either 1 or 0. No leading or trailing quotation characters are present.

-- The representation of a BOOLEAN value is formed by an identifier, either FALSE or TRUE.

-- The representation of a CHARACTER value is formed by a single character.

-- The representation of both INTEGER and REAL values is that of a decimal literal (see 13.4.1), with the addition of an optional leading sign. The sign is never written if the value is nonnegative, but it is accepted during a read even if the value is nonnegative. No spaces can occur between the sign and the remainder of the value. The decimal point is absent in the case of an INTEGER literal and present in the case of a REAL literal. An exponent may optionally be present; moreover, the language does not define under what conditions it is or is not present. However, if the exponent is present, the "e" is written as a lowercase character. Leading and trailing zeroes are written as necessary to meet the requirements of the FIELD and DIGITS parameters, and they are accepted during a read.

-- The representation of a STRING value is formed by a sequence of characters, one for each element of the string. No leading or trailing quotation characters are present.

-- The representation of a TIME value is formed by an optional decimal literal composed following the rules for INTEGER and REAL literals described above, one or more blanks, and an identifier that is a unit of type TIME, as defined in package STANDARD (see 14.2). When read, the identifier can be expressed with characters of either case; when written, the identifier is expressed in lowercase characters.

Each WRITE procedure similarly appends data to the end of the string value designated by parameter L; in this case, however, L continues to designate the entire line after the value is appended. The format of the appended data is defined by the string representations defined above for the READ procedures.

The READ and WRITE procedures for the types BIT_VECTOR and STRING respectively read and write the element values in left-to-right order.

For each predefined data type there are two READ procedures declared in package TEXTIO. The first has three parameters: L, the line to read from; VALUE, the value read from the line; and GOOD, a Boolean flag that indicates whether the read operation succeeded or not. For example, the operation READ (L, IntVal,OK) would return with OK set to FALSE, L unchanged, and IntVal undefined if IntVal is a variable of type INTEGER and L designates the line "ABC". The success indication returned via parameter GOOD allows a process to recover gracefully from unexpected discrepancies in input format. The second form of read operation has only the parameters L and VALUE. If the requested type cannot be read into VALUE from line L, then an error occurs. Thus, the operation READ (L, IntVal) would cause an error to occur if IntVal is of type INTEGER and L designates the line "ABC".

For each predefined data type there is one WRITE procedure declared in package TEXTIO. Each of these has at least two parameters: L, the line to which to write; and VALUE, the value to be written. The additional parameters JUSTIFIED, FIELD, DIGITS, and UNIT control the formatting of output data. Each write operation appends data to a line formatted within a *field* that is at least as long as required to represent the data value. Parameter FIELD specifies the desired field width. Since the actual field width will always beat least large enough to hold the string representation of the data value, the default value 0 for the FIELD parameter has the effect of causing the data value to be written out in a field of exactly the right width (i.e., no leading or trailing spaces). Parameter JUSTIFIED specifies whether values are to be right- or left-justified within the field; the default is right-justified. If the FIELD parameter describes a field width larger than the number of characters necessary for a given value, blanks are used to fill the remaining characters in the field.

Parameter DIGITS specifies how many digits to the right of the decimal point are to be output when writing a real number; the default value 0 indicates that the number should be output in standard form, consisting of a normalized mantissa plus exponent (e.g. 1.079236E-23). If DIGITS is nonzero, then the real number is output as an integer part followed by '.' followed by the fractional part, using the specified number of digits (e.g., 3.14159).

Parameter UNIT specifies how values of type TIME are to be formatted. The value of this parameter must be equal to one of the units declared as part of the declaration of type TIME; the result is that the TIME value is formatted as an integer or real literal representing the number of multiples of this unit,followed by the name of the unit itself. The name of the unit is formatted using only lowercase characters. Thus the procedure call WRITE(Line, 5 ns,UNIT=>us) would result in the string value " 0.005 us" being appended to the string value designated by Line, whereas WRITE(Line, 5 ns) would result in the string value "5 ns" being appended (since the default UNIT value is ns).

Function ENDFILE is defined for files of type TEXT by the implicit declaration of that function as part of the declaration of the file type.

NOTES

1--For a variable L of type Line, attribute L'Length gives the current length of the line, whether that line is being read or written. For a line L that is being written, the value of L'Length gives the number of characters that have already been written to the line; this is equivalent to the column number of the last character of the line. For a line L that is being read, the value of L'Length gives the number of characters on that line remaining to be read. In particular, the expression L'Length = 0 is true precisely when the end of the current line has been reached.

2--The execution of a read or write operation may modify or even deallocate the string object designated by input parameter L of type Line for that operation; thus, a dangling reference may result if the value of a variable L of type Line is assigned to another access variable and then a read or write operation is performed on L.

Section 7

Expressions

The rules applicable to the different forms of expression, and to their evaluation, are given in this section.

7.1 Expressions

An expression is a formula that defines the computation of a value.

```
expression ::=
      relation { and relation }
   | relation { or relation }
   | relation { xor relation }
   | relation [ nand relation ]
   | relation [ nor relation ]
   | relation { xnor relation }

relation ::=
    shift_expression [ relational_operator shift_expression ]

shift_expression ::=
    simple_expression [ shift_operator simple_expression ]

simple_expression ::=
    [ sign ] term { adding_operator term }

term ::=
    factor { multiplying_operator factor }

factor ::=
      primary [ ** primary ]
   | abs primary
   | not primary

primary ::=
      name
   | literal
   | aggregate
   | function_call
   | qualified_expression
   | type_conversion
   | allocator
   | ( expression )
```

Each primary has a value and a type. The only names allowed as primaries are attributes that yield values and names denoting objects or values. In the case of names denoting objects, the value of the primary is the value of the object.

The type of an expression depends only upon the types of its operands and on the operators applied; for an overloaded operand

or operator, the determination of the operand type, or the identification of the overloaded operator, depends on the context (see 10.5). For each predefined operator, the operand and result types are given in the following clause.

NOTE--The syntax for an expression involving logical operators allows a sequence of **and**, **or**, **xor**, or **xnor** operators (whether predefined or user-defined), since the corresponding predefined operations are associative. For the operators **nand** and **nor** (whether predefined or user-defined), however, such a sequence is not allowed, since the corresponding predefined operations are not associative.

7.2 Operators

The operators that may be used in expressions are defined below. Each operator belongs to a class of operators, all of which have the same precedence level;the classes of operators are listed in order of increasing precedence.

logical_operator	::=	**and**	|	**or**	|	**nand**	|	**nor**	|	**xor**	|	**xnor**
relational_operator	::=	=	|	/=	|	<	|	<=	|	>	|	>=
shift_operator	::=	**sll**	|	**srl**	|	**sla**	|	**sra**	|	**rol**	|	**ror**
adding_operator	::=	+	|	-	|	&						
sign	::=	+	|	-								
mutiplying_operator	::=	*	|	/	|	**mod**	|	**rem**				
micellaneous_operator	::=	**	|	**abs**	|	**not**						

Operators of higher precedence are associated with their operands before operators of lower precedence. Where the language allows a sequence of operators, operators with the same precedence level are associated with their operands in textual order, from left to right. The precedence of an operator is fixed and may not be changed by the user, but parentheses can be used to control the association of operators and operands.

In general, operands in an expression are evaluated before being associated with operators. For certain operations, however, the right-hand operand is evaluated if and only if the left-hand operand has a certain value. These operations are called *short-circuit* operations. The logical operations **and**, **or**, **nand**, and **nor** defined for operands of types BIT and BOOLEAN are all short-circuit operations; furthermore, these are the only short-circuit operations.

Every predefined operator is a pure function (see 2.1). No predefined operators have named formal parameters; therefore, named association (see 4.3.2.2) may not be used when invoking a predefined operation.

NOTES

1--The predefined operators for the standard types are declared in package STANDARD as shown in 14.2 .

2--The operator **not** is classified as a miscellaneous operator for the purposes of defining precedence, but is otherwise classified as a logical operator.

7.2.1 Logical operators

The logical operators **and**, **or**, **nand**, **nor**,**xor**, **xnor**, and **not** are defined for predefined types BIT and BOOLEAN. They are also defined for any one-dimensional array type whose element type is BIT or BOOLEAN. For the binary operators **and**,**or**, **nand**, **nor**, **xor**, and **xnor**, the operands must be of the same base type. Moreover, for the binary operators **and**,**or**, **nand**, **nor**, **xor**, and **xnor** defined on one-dimensional array types, the operands must be arrays of the same length,the operation is performed on matching elements of the arrays, and the result is an array with the same index range as the left operand. For the unary operator **not** defined on one-dimensional array types, the operation is performed on each element of the operand, and the result is an array with the same index range as the operand.

The effects of the logical operators are defined in the following tables. The symbol T represents TRUE for type BOOLEAN, '1' for type BIT; the symbol F represents FALSE for type BOOLEAN, '0' for type BIT.

A	B	A **and** B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A **or** B
T	T	T
T	F	T
F	T	T
F	F	F

A	B	A **xor** B
T	T	F
T	F	T
F	T	T
F	F	T

A	B	A **nand** B
T	F	F
T	T	T
F	T	T
F	T	T

A	B	A **nor** B
T	T	F
T	F	F
F	T	F
F	F	T

A	B	A **xnor** B
T	T	T
T	F	F
F	T	F
F	F	T

A	**not A**
T	F
F	T

For the short-circuit operations **and**, **or**, **nand**, and **nor** on types BIT and BOOLEAN, the right operand is evaluated only if the value of the left operand is not sufficient to determine the result of the operation. For operations **and** and **nand**, the right operand is evaluated only if the value of the left operand is T; for operations **or** and **nor**, the right operand is evaluated only if the value of the left operand is F.

NOTE--All of the binary logical operators belong to the class of operators with the lowest precedence. The unary logical operator **not** belongs to the class of operators with the highest precedence.

7.2.2 Relational operators

Relational operators include tests for equality, inequality, and ordering of operands. The operands of each relational operator must be of the same type. The result type of each relational operator is the predefined type BOOLEAN.

Operator	Operation	Operand type	Result type
=	Equality	Any type	BOOLEAN
/=	Inequality	Any type	BOOLEAN
< <= > >=	Ordering	Any scalar type or discrete array type	BOOLEAN

The equality and inequality operators (= and /=) are defined for all types other than file types. The equality operator returns the value TRUE if the two operands are equal and returns the value FALSE otherwise. The inequality operator returns the value FALSE if the two operands are equal and returns the value TRUE otherwise.

Two scalar values of the same type are equal if and only if the values are the same. Two composite values of the same type are equal if and only if for each element of the left operand there is a *matching element* of the right operand and vice versa, and the values of matching elements are equal, as given by the predefined equality operator for the element type. In particular, two null arrays of the same type are always equal. Two values of an access type are equal if and only if they both designate the same

object or they both are equal to the null value for the access type.

For two record values, matching elements are those that have the same element identifier. For two one-dimensional array values, matching elements are those (if any) whose index values match in the following sense: the left bounds of the index ranges are defined to match; if two elements match, the elements immediately to their right are also defined to match. For two multi-dimensional array values, matching elements are those whose indices match in successive positions.

The ordering operators are defined for any scalar type and for any discrete array type. A *discrete array* is a one-dimensional array whose elements are of a discrete type. Each operator returns TRUE if the corresponding relation is satisfied; otherwise, the operator returns FALSE.

For scalar types, ordering is defined in terms of the relative values. For discrete array types, the relation < (less than) is defined such that the left operand is less than the right operand if and only if

-- The left operand is a null array and the right operand is a nonnull array; otherwise,

-- Both operands are nonnull arrays, and one of the following conditions is satisfied:

-- The leftmost element of the left operand is less than that of the right;or

-- The leftmost element of the left operand is equal to that of the right, and the tail of the left operand is less than that of the right (the tail consists of the remaining elements to the right of the leftmost element and can be null).

The relation <= (less than or equal) for discrete array types is defined to be the inclusive disjunction of the results of the < and = operators for the same two operands. The relations > (greater than) and >= (greater than or equal) are defined to be the complements of the <= and <operators respectively for the same two operands.

7.2.3 Shift operators

The shift operators **sll**, **srl**, **sla**, **sra**, **rol**,and **ror** are defined for any one-dimensional array type whose element type is either of the predefined types BIT or BOOLEAN.

Operator	Operation	Left operand type	Right operand type	Result type
sll	Shift left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
srl	Shift right logical	Any one-dimensional array type whose element typeis BIT or BOOLEAN	INTEGER	Same as left
sla	Shift left arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sra	Shift right arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
rol	Rotate left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
ror	Rotate right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left

The index subtypes of the return values of all shift operators are the same as the index subtypes of their left arguments.

The values returned by the shift operators are defined as follows. In the remainder of this section, the values of their leftmost arguments are referred to as L and the values of their rightmost arguments are referred to as R.

-- The **sll** operator returns a value that is L logically shifted left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'Length -1) elements of L and whose right argument is T'Left, where T is the element type of L. If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **srl** -R.

-- The **srl** operator returns a value that is L logically shifted right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'Length -1) elements of L and whose left argument is T'Left, where T is the element type of L. If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sll** -R.

-- The **sla** operator returns a value that is L arithmetically shifted left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost(L'Length - 1) elements of L and whose right argument is L(L'Right). If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **sra** -R.

-- The **sra** operator returns a value that is L arithmetically shifted right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic shift operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost(L'Length - 1) elements of L and whose left argument is L(L'Left). If R is positive, this basic shift operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L**sla** -R.

-- The **rol** operator returns a value that is L rotated left by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic rotate operation replaces L with a value that is the result of a concatenation whose left argument is the rightmost (L'Length - 1) elements of L and whose right argument is L(L'Left). If R is positive, this basic rotate operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **ror** -R.

-- The **ror** operator returns a value that is L rotated right by R index positions. That is, if R is 0 or if L is a null array, the return value is L. Otherwise, a basic rotate operation replaces L with a value that is the result of a concatenation whose right argument is the leftmost (L'Length - 1) elements of L and whose left argument is L(L'Right). If R is positive, this basic rotate operation is repeated R times to form the result. If R is negative, then the return value is the value of the expression L **rol** -R.

NOTES

1--The logical operators may be overloaded, for example, to disallow negative integers as the second argument.

2--The subtype of the result of a shift operator is the same as that of the left operand.

7.2.4 Adding operators

The adding operators + and - are predefined for any numeric type and have their conventional mathematical meaning. The concatenation operator & is predefined for any one-dimensional array type.

Operator	Operation	Left operand type	Right operand type	Result type
+	Addition	Any numeric type	Same type	Same type
-	Subtraction	Any numeric type	Same type	Same type
&	Concatenation	Any array type	Same array type	Same array type
		Any array type	The element type	Same array type

The element type	Any array type	Same array type
The element type	The element type	Any array type

For concatenation, there are three mutually exclusive cases:

a. If both operands are one-dimensional arrays of the same type, the result of the concatenation is a one-dimensional array of this same type whose length is the sum of the lengths of its operands, and whose elements consist of the elements of the left operand (in left-to-right order) followed by the elements of the right operand (in left-to-right order). The direction of the result is the direction of the left operand, unless the left operand is a null array, in which case the direction of the result is that of the right operand.

   If both operands are null arrays, then the result of the concatenation is the right operand. Otherwise, the direction and bounds of the result are determined as follows: Let S be the index subtype of the base type of the result. The direction of the result of the concatenation is the direction of S, and the left bound of the result is S'LEFT.

b. If one of the operands is a one-dimensional array and the type of the other operand is the element type of this aforementioned one-dimensional array, the result of the concatenation is given by the rules in case 1, using in place of the other operand an implicit array having this operand as its only element.

c. If both operands are of the same type and it is the element type of some one-dimensional array type, the type of the result must be known from the context and is this one-dimensional array type. In this case, each operand is treated as the one element of an implicit array, and the result of the concatenation is determined as in case a.

In all cases, it is an error if either bound of the index subtype of the result does not belong to the index subtype of the type of the result, unless the result is a null array. It is also an error if any element of the result does not belong to the element subtype of the type of the result.

Examples:

```
     subtype BYTE is BIT_VECTOR (7 downto 0);
     type MEMORY is array (Natural range <>) of BYTE;

     --  The following concatenation accepts two BIT_VECTORs and returns a BIT_VECTOR
     --  (case a):

     constant ZERO: BYTE := "0000" & "0000";

     --  The next two examples show that the same expression can represent either
case a or
     --  case c, depending on the context of the expression.

     --  The following concatenation accepts two BIT_VECTORS and returns a BIT_VECTOR
     --  (case a):


     constant C1: BIT_VECTOR := ZERO & ZERO;

     --  The following concatenation accepts two BIT_VECTORs and returns a MEMORY
     --  (case c):

     constant C2: MEMORY := ZERO & ZERO;

     --  The following concatenation accepts a BIT_VECTOR and a MEMORY, returning a
     --  MEMORY (case b):

     constant C3: MEMORY := ZERO & C2;
```

```
     --  The following concatenation accepts a MEMORY and a BIT_VECTOR, returning a
     --  MEMORY (case b):

     constant C4: MEMORY := C2 & ZERO;

     --  The following concatenation accepts two MEMORYs and returns a MEMORY (case
a):

     constant C5: MEMORY := C2 & C3;

     type R1 is 0 to 7;
     type R2 is 7 downto 0;

     type T1 is array (R1 range <>) of Bit;
     type T2 is array (R2 range <>) of Bit;

     subtype S1 is T1(R1);
     subtype S2 is T2(R2);

     constant K1: S1 := (others => '0');
     constant K2: T1 := K1(1 to 3) & K1(3 to 4);              --  K2'Left = 0 and
K2'Right = 4
     constant K3: T1 := K1(5 to 7) & K1(1 to 2);              --  K3'Left = 0 and
K3'Right = 4
     constant K4: T1 := K1(2 to 1) & K1(1 to 2);              --  K4'Left = 0 and
K4'Right = 1

     constant K5: S2 := (others => '0');
     constant K6: T2 := K5(3 downto 1) & K5(4 downto 3);      --  K6'Left = 7 and
K6'Right = 3
     constant K7: T2 := K5(7 downto 5) & K5(2 downto 1)       --  K7'Left = 7 and
K7'Right = 3
     constant K8: T2 := K5(1 downto 2) & K5(2 downto 1);      --  K8'Left = 7 and
K8'Right = 6
```

NOTES

1--For a given concatenation whose operands are of the same type, there maybe visible more than one array type that could be the result type according to the rules of case 3. The concatenation is ambiguous and therefore an error if, using the overload resolution rules of 2.3 and 10.5 , the type of the result is not uniquely determined.

2--Additionally, for a given concatenation, there may be visible array types that allow both case a and case c to apply. The concatenation is again ambiguous and therefore an error if the overload resolution rules cannot be used to determine a result type uniquely.

7.2.5 Sign operators

Signs + and - are predefined for any numeric type and have their conventional mathematical meaning: they respectively represent the identity and negation functions. For each of these unary operators, the operand and the result have the same type.

Operator	Operation	Operand type	Result type
+	Identity	Any numeric type	Same type
-	Negation	Any numeric type	Same type

NOTE--Because of the relative precedence of signs + and - in the grammar for expressions, a signed operand must not follow a multiplying operator, the exponentiating operator **, or the operators **abs** and **not**. For example, the syntax does not allow the following expressions:

```
A/+B            --  An illegal expression
A**-B           --  An illegal expression
```

However,these expressions may be rewritten legally as follows:

```
A/(+B)          --  A legal expression
A**(-B)         --  A legal expression
```

7.2.6 Multiplying operators

The operators * and / are predefined for any integer and any floating point type and have their conventional mathematical meaning; the operators **mod** and **rem** are predefined for any integer type. For each of these operators, the operands and the result are of the same type.

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	Any integer type	Same type	Same type
		Any floating point type	Same type	Same type
/	Division	Any integer type	Same type	Same type
		Any floating point type	Same type	Same type
mod	Modulus	Any integer type	Same type	Same type
rem	Remainder	Any integer type	Same type	Same type

Integer division and remainder are defined by the following relation:

```
A = (A/B)*B + (A rem B)
```

where(A **rem** B) has the sign of A and an absolute value less than the absolute value of B. Integer division satisfies the following identity:

```
(-A)/B = -(A/B) = A/(-B)
```

The result of the modulus operation is such that (A **mod** B) has the sign of B and an absolute value less than the absolute value of B; in addition, for some integer value N, this result must satisfy the relation:

```
A = B*N + (A mod B)
```

In addition to the above table, the operators * and / are predefined for any physical type.

Operator	Operation	Left operand type	Right operand type	Result type
*	Multiplication	Any physical type	INTEGER	Same as left
		Any physical type	REAL	Same as left
		INTEGER	Any physical type	Same as right

		REAL	Any physical type	Same as right
/	Division	Any physical type	INTEGER	Same as left
		Any physical type	REAL	Same as left
		Any physical type	The same type	*Universal integer*

Multiplication of a value P of a physical type T_p by a value I of type INTEGER is equivalent to the following computation:

```
Tp'Val( Tp'Pos(P) * I )
```

Multiplication of a value P of a physical type T_p by a value F of type REAL is equivalent to the following computation:

```
Tp'Val( INTEGER( REAL( Tp'Pos(P) ) * F ))
```

Division of a value P of a physical type T_p by a value I of type INTEGER is equivalent to the following computation:

```
Tp'Val( Tp'Pos(P) / I )
```

Division of a value P of a physical type T_p by a value F of type REAL is equivalent to the following computation:

```
Tp'Val( INTEGER( REAL( Tp'Pos(P) ) / F ))
```

Division of a value P of a physical type T_p by a value P2 of the same physical type is equivalent to the following computation:

```
Tp'Pos(P) / Tp'Pos(P2)
```

Examples:

```
  5   rem   3  =  2
  5   mod   3  =  2

(-5)  rem   3  = -2
(-5)  mod   3  =  1

(-5)  rem (-3) = -2
(-5)  mod (-3) = -2

  5   rem (-3) =  2
  5   mod (-3) = -1
```

NOTE--Because of the precedence rules (see 7.2), the expression "-5 **rem** 2" is interpreted as "-(5 **rem** 2)" and not as "(-5) **rem** 2".

7.2.7 Miscellaneous operators

The unary operator **abs** is predefined for any numeric type.

Operator	Operation	Operand type	Result type
abs	Absolute value	Any numeric type	Same numeric type

The *exponentiating* operator ** is predefined for each integer type and for each floating point type. In either case the right operand, called the exponent, is of the predefined type INTEGER.

Operator	Operation	Left operand type	Right operand type	Result type
**	Exponentiation	Any integer type	INTEGER	Same as left
		Any floating point type	INTEGER	Same as left

Exponentiation with an integer exponent is equivalent to repeated multiplication of the left operand by itself for a number of times indicated by the absolute value of the exponent and from left to right; if the exponent is negative, then the result is the reciprocal of that obtained with the absolute value of the exponent. Exponentiation with a negative exponent is only allowed for a left operand of a floating point type. Exponentiation by a zero exponent results in the value one. Exponentiation of a value of a floating point type is approximate.

7.3 Operands

The operands in an expression include names (that denote objects, values, or attributes that result in a value), literals, aggregates, function calls, qualified expressions, type conversions, and allocators. In addition, an expression enclosed in parentheses may be an operand in an expression. Names are defined in 6.1 ; the other kinds of operands are defined in the following subclauses.

7.3.1 Literals

A literal is either a numeric literal, an enumeration literal, a string literal, a bit string literal, or the literal **null**.

```
literal ::=
        numeric_literal
      | enumeration_literal
      | string_literal
      | bit_string_literal
      | null

numeric_literal ::=
        abstract_literal
      | physical_literal
```

Numeric literals include literals of the abstract types *universal_integer* and *universal_real*, as well as literals of physical types. Abstract literals are defined in 13.4 ; physical literals are defined in 3.1.3 .

Enumeration literals are literals of enumeration types. They include both identifiers and character literals. Enumeration literals are defined in 3.1.1 .

String and bit string literals are representations of one-dimensional arrays of characters. The type of a string or bit string literal must be determinable solely from the context in which the literal appears, excluding the literal itself but using the fact that the type of the literal must be a one-dimensional array of a character type. The lexical structure of string and bit string literals is defined in Section 13, Lexical Elements.

For a nonnull array value represented by either a string or bit string literal,the direction and bounds of the array value are determined according to the rules for positional array aggregates, where the number of elements in the aggregate is equal to the length (see 13.6 and 13.7) of the string or bit string literal. For a null array value represented by either a string or bit string literal, the direction and leftmost bound of the array value are determined as in the non-null case. If the direction is ascending, then the rightmost bound is the predecessor (as given by the 'PRED attribute) of the leftmost bound; otherwise the rightmost

bound is the successor (as given by the 'SUCC attribute) of the leftmost bound.

The character literals corresponding to the graphic characters contained within a string literal or a bit string literal must be visible at the place of the string literal.

The literal **null** represents the null access value for any access type.

Evaluation of a literal yields the corresponding value.

Examples:

```
3.14159_26536          --  A literal of type universal_real.
5280                   --  A literal of type universal_integer.
10.7 ns                --  A literal of a physical type.
O"4777"                --  A bit-string literal.
"54LS281"              --  A string literal.
""                     --  A string literal representing a null array.
```

7.3.2 Aggregates

An aggregate is a basic operation (see the introduction to Section 3) that combines one or more values into a composite value of a record or array type.

```
aggregate ::=
    ( element_association { , element_association } )

element_association ::=
    [ choices => ] expression

choices ::=  choice { | choice }


choice ::=
       simple_expression
    | discrete_range
    | element_simple_name
    | others
```

Each element association associates an expression with elements (possibly none). An element association is said to be *named* if the elements are specified explicitly by choices; otherwise, it is said to be *positional*. For a positional association, each element is implicitly specified by position in the textual order of the elements in the corresponding type declaration.

Both named and positional associations can be used in the same aggregate, with all positional associations appearing first (in textual order) and all named associations appearing next (in any order, except that no associations may follow an **others** association). Aggregates containing a single element association must always be specified using named association in order to distinguish them from parenthesized expressions.

An element association with a choice that is an element simple name is only allowed in a record aggregate. An element association with a choice that is a simple expression or a discrete range is only allowed in an array aggregate: a simple expression specifies the element at the corresponding index value, whereas a discrete range specifies the elements at each of the index values in the range. The discrete range has no significance other than to define the set of choices implied by the discrete range. In particular, the direction specified or implied by the discrete range has no significance. An element association with the choice **others** is allowed in either an array aggregate or a record aggregate if the association appears last and has this single choice; it specifies all remaining elements, if any.

Each element of the value defined by an aggregate must be represented once and only once in the aggregate.

The type of an aggregate must be determinable solely from the context in which the aggregate appears, excluding the aggregate itself but using the fact that the type of the aggregate must be a composite type. The type of an aggregate in turn determines the required type for each of its elements.

7.3.2.1 Record aggregates

If the type of an aggregate is a record type, the element names given as choices must denote elements of that record type. If the choice **others** is given as a choice of a record aggregate, it must represent at least one element. An element association with more than one choice, or with the choice **others**, is only allowed if the elements specified are all of the same type. The expression of an element association must have the type of the associated record elements.

A record aggregate is evaluated as follows. The expressions given in the element associations are evaluated in an order (or lack thereof) not defined by the language. The expression of a named association is evaluated once for each associated element. A check is made that the value of each element of the aggregate belongs to the subtype of this element. It is an error if this check fails.

7.3.2.2 Array aggregates

For an aggregate of a one-dimensional array type, each choice must specify values of the index type, and the expression of each element association must be of the element type. An aggregate of an n-dimensional array type, where n is greater than 1, is written as a one-dimensional aggregate in which the index subtype of the aggregate is given by the first index position of the array type, and the expression specified for each element association is an (n-1)-dimensional array or array aggregate, which is called a *subaggregate*. A string or bit string literal is allowed as a subaggregate in the place of any aggregate of a one-dimensional array of a character type.

Apart from a final element association with the single choice **others**,the rest (if any) of the element associations of an array aggregate must be either all positional or all named. A named association of an array aggregate is allowed to have a choice that is not locally static, or likewise a choice that is a null range, only if the aggregate includes a single element association and this element association has a single choice. An **others** choice is locally static if the applicable index constraint is locally static.

The subtype of an array aggregate that has an **others** choice must be determinable from the context. That is, an array aggregate with an **others** choice may only appear

a. As an actual associated with a formal parameter or formal generic declared to be of a constrained array subtype (or subelement thereof)
b. As the default expression defining the default initial value of a port declared to be of a constrained array subtype
c. As the result expression of a function, where the corresponding function result type is a constrained array subtype
d. As a value expression in an assignment statement, where the target is a declared object, and the subtype of the target is a constrained array subtype (or subelement of such a declared object)
e. As the expression defining the initial value of a constant or variable object, where that object is declared to be of a constrained array subtype
f. As the expression defining the default values of signals in a signal declaration, where the corresponding subtype is a constrained array subtype
g. As the expression defining the value of an attribute in an attribute specification, where that attribute is declared to be of a constrained array subtype
h. As the operand of a qualified expression whose type mark denotes a constrained array subtype
i. i) As a subaggregate nested within an aggregate, where that aggregate itself appears in one of these contexts

The bounds of an array that does not have an **others** choice are determined as follows. If the aggregate appears in one of the contexts in the preceding list, then the direction of the index subtype of the aggregate is that of the corresponding constrained array subtype; otherwise, the direction of the index subtype of the aggregate is that of the index subtype of the base type of the aggregate. For an aggregate that has named associations, the leftmost and rightmost bounds are determined by the direction of

the index subtype of the aggregate and the smallest and largest choices given. For a positional aggregate, the leftmost bound is determined by the applicable index constraint if the aggregate appears in one of the contexts in the preceding list; otherwise, the leftmost bound is given by S'LEFT where S is the index subtype of the base type of the array. In either case, the rightmost bound is determined by the direction of the index subtype and the number of elements.

The evaluation of an array aggregate that is not a subaggregate proceeds in two steps. First, the choices of this aggregate and of its subaggregates, if any,are evaluated in some order (or lack thereof) that is not defined by the language. Second, the expressions of the element associations of the array aggregate are evaluated in some order that is not defined by the language; the expression of a named association is evaluated once for each associated element. The evaluation of a subaggregate consists of this second step (the first step is omitted since the choices have already been evaluated).

For the evaluation of an aggregate that is not a null array, a check is made that the index values defined by choices belong to the corresponding index subtypes, and also that the value of each element of the aggregate belongs to the subtype of this element. For a multidimensional aggregate of dimension n, a check is made that all (n-1)-dimensional subaggregates have the same bounds. It is an error if any one of these checks fails.

7.3.3 Function calls

A function call invokes the execution of a function body. The call specifies the name of the function to be invoked and specifies the actual parameters, if any, to be associated with the formal parameters of the function. Execution of the function body results in a value of the type declared to be the result type in the declaration of the invoked function.

```
function_call ::=
   function_name [ ( actual_parameter_part ) ]

actual_parameter_part ::=  parameter_association_list
```

For each formal parameter of a function, a function call must specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element (other than the actual part **open**) in the association list, or in the absence of such an association element, by a default expression (see 4.3.2).

Evaluation of a function call includes evaluation of the actual parameter expressions specified in the call and evaluation of the default expressions associated with formal parameters of the function that do not have actual parameters associated with them. In both cases, the resulting value must belong to the subtype of the associated formal parameter. (If the formal parameter is of an unconstrained array type, then the formal parameter takes on the subtype of the actual parameter.) The function body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

NOTE--If a name (including one used as a prefix) has an interpretation both as a function call and an indexed name, then the innermost complete context is used to disambiguate the name. If, after applying this rule, there is not exactly one interpretation of the name, then the name is ambiguous. See 10.5 .

7.3.4 Qualified expressions

A qualified expression is a basic operation (see the introduction to Section 3)that is used to explicitly state the type, and possibly the subtype, of an operand that is an expression or an aggregate.

```
qualified_expression ::=
       type_mark ' ( expression )
     | type_mark ' aggregate
```

The operand must have the same type as the base type of the type mark. The value of a qualified expression is the value of the operand. The evaluation of a qualified expression evaluates the operand and checks that its value belongs to the subtype denoted by the type mark.

NOTE--Whenever the type of an enumeration literal or aggregate is not known from the context, a qualified expression can be used to state the type explicitly.

7.3.5 Type conversions

A type conversion provides for explicit conversion between closely related types.

```
type_conversion ::=  type_mark ( expression )
```

The target type of a type conversion is the base type of the type mark. The type of the operand of a type conversion must be determinable independent of the context (in particular, independent of the target type). Furthermore, the operand of a type conversion is not allowed to be the literal **null**, an allocator, an aggregate, or a string literal. An expression enclosed by parentheses is allowed as the operand of a type conversion only if the expression alone is allowed.

If the type mark denotes a subtype, conversion consists of conversion to the target type followed by a check that the result of the conversion belongs to the subtype.

Explicit type conversions are allowed between *closely related types*. In particular, a type is closely related to itself. Other types are closely related only under the following conditions:

a. *Abstract Numeric Types*--Any abstract numeric type is closely related to any other abstract numeric type. In an explicit type conversion where the type mark denotes an abstract numeric type, the operand can be of any integer or floating point type. The value of the operand is converted to the target type, which must also be an integer or floating point type. The conversion of a floating point value to an integer type rounds to the nearest integer; if the value is halfway between two integers, rounding may be up or down.
b. *Array Types*--Two array types are closely related if and only if

   -- The types have the same dimensionality;

   -- For each index position, the index types are either the same or are closely related; and

   -- The element types are the same.

   In an explicit type conversion where the type mark denotes an array type, the following rules apply: if the type mark denotes an unconstrained array type and if the operand is not a null array, then, for each index position, the bounds of the result are obtained by converting the bounds of the operand to the corresponding index type of the target type. If the type mark denotes a constrained array subtype, then the bounds of the result are those imposed by the type mark. In either case, the value of each element of the result is that of the matching element of the operand (see 7.2.2).

No other types are closely related.

In the case of conversions between numeric types, it is an error if the result of the conversion fails to satisfy a constraint imposed by the type mark.

In the case of conversions between array types, a check is made that any constraint on the element subtype is the same for the operand array type as for the target array type. If the type mark denotes an unconstrained array type, then, for each index position, a check is made that the bounds of the result belong to the corresponding index subtype of the target type. If the type mark denotes a constrained array subtype, a check is made that for each element of the operand there is a matching element of the target subtype, and vice versa. It is an error if any of these checks fail.

In certain cases, an implicit type conversion will be performed. An implicit conversion of an operand of type *universal_integer* to another integer type, or of an operand of type *universal_real* to another floating point type, can only be applied if the operand is either a numeric literal or an attribute, or if the operand is an expression consisting of the division of a value of a

physical type by a value of the same type; such an operand is called a *convertible* universal operand. An implicit conversion of a convertible universal operand is applied if and only if the innermost complete context determines a unique (numeric) target type for the implicit conversion, and there is no legal interpretation of this context without this conversion.

NOTE--Two array types may be closely related even if corresponding index positions have different directions.

7.3.6 Allocators

The evaluation of an allocator creates an object and yields an access value that designates the object.

```
allocator ::=
        new subtype_indication
      | new qualified_expression
```

The type of the object created by an allocator is the base type of the type mark given in either the subtype indication or the qualified expression. For an allocator with a subtype indication, the initial value of the created object is the same as the default initial value for an explicitly declared variable of the designated subtype. For an allocator with a qualified expression, this expression defines the initial value of the created object.

The type of the access value returned by an allocator must be determinable solely from the context, but using the fact that the value returned is of an access type having the named designated type.

The only allowed form of constraint in the subtype indication of an allocator is an index constraint. If an allocator includes a subtype indication and if the type of the object created is an array type, then the subtype indication must either denote a constrained subtype or include an explicit index constraint. A subtype indication that is part of an allocator must not include a resolution function.

If the type of the created object is an array type, then the created object is always constrained. If the allocator includes a subtype indication, the created object is constrained by the subtype. If the allocator includes a qualified expression, the created object is constrained by the bounds of the initial value defined by that expression. For other types, the subtype of the created object is the subtype defined by the subtype of the access type definition.

For the evaluation of an allocator, the elaboration of the subtype indication or the evaluation of the qualified expression is first performed. The new object is then created, and the object is then assigned its initial value. Finally, an access value that designates the created object is returned.

In the absence of explicit deallocation, an implementation must guarantee that any object created by the evaluation of an allocator remains allocated for as long as this object or one of its subelements is accessible directly or indirectly; that is, as long as it can be denoted by some name.

NOTES

1--Procedure Deallocate is implicitly declared for each access type. This procedure provides a mechanism for explicitly deallocating the storage occupied by an object created by an allocator.

2--An implementation may (but need not) deallocate the storage occupied by an object created by an allocator, once this object has become inaccessible.

Examples:

```
     new NODE  --  Takes on default initial value.
     new NODE'(15 ns, null)                              --  Initial value is
specified.
     new NODE'(Delay => 5  ns, \Next\ => Stack)          --  Initial value is
```

```
specified.
     new BIT_VECTOR'("00110110")                    --  Constrained by initial
value.
     new STRING (1 to 10)                           --  Constrained by index
constraint.
     new STRING                                     --  Illegal: must be
constrained.
```

7.4 Static expressions

Certain expressions are said to be *static*. Similarly, certain discrete ranges are said to be static, and the type marks of certain subtypes are said to denote static subtypes.

There are two categories of static expression. Certain forms of expression can be evaluated during the analysis of the design unit in which they appear; such an expression is said to be *locally static*. Certain forms of expression can be evaluated as soon as the design hierarchy in which they appear is elaborated; such an expression is said to be *globally static*.

7.4.1 Locally static primaries

An expression is said to be locally static if and only if every operator in the expression denotes an implicitly defined operator whose operands and result are scalar and if every primary in the expression is a *locally static primary*, where a locally static primary is defined to be one of the following:

a. A literal of any type other than type TIME
b. A constant (other than a deferred constant) explicitly declared by a constant declaration and initialized with a locally static expression
c. An alias whose aliased name (given in the corresponding alias declaration) is a locally static primary
d. A function call whose function name denotes an implicitly defined operator, and whose actual parameters are each locally static expressions
e. A predefined attribute that is a value, other than the predefined attribute 'PATH_NAME, and whose prefix is either a locally static subtype or is an object name that is of a locally static subtype
f. A predefined attribute that is a function, other than the predefined attributes 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE, whose prefix is either a locally static subtype or is an object that is of a locally static subtype, and whose actual parameter (if any)is a locally static expression
g. A user-defined attribute whose value is defined by a locally static expression
h. A qualified expression whose operand is a locally static expression
i. A type conversion whose expression is a locally static expression
j. A locally static expression enclosed in parentheses

A locally static range is either a range of the second form (see 3.1) whose bounds are locally static expressions, or a range of the first form whose prefix denotes either a locally static subtype or an object that is of a locally static subtype. A locally static range constraint is a range constraint whose range is locally static. A locally static scalar subtype is either a scalar base type or a scalar subtype formed by imposing on a locally static subtype a locally static range constraint. A locally static discrete range is either a locally static subtype or a locally static range.

A locally static index constraint is an index constraint for which each index subtype of the corresponding array type is locally static and in which each discrete range is locally static. A locally static array subtype is a constrained array subtype formed by imposing on an unconstrained array type a locally static index constraint. A locally static record subtype is a record type whose fields are all of locally static subtypes. A locally static access subtype is a subtype denoting an access type. A locally static file subtype is a subtype denoting a file type.

A locally static subtype is either a locally static scalar subtype, a locally static array subtype, a locally static record subtype, a locally static access subtype, or a locally static file subtype.

7.4.2 Globally static primaries

An expression is said to be globally static if and only if every operator in the expression denotes a pure function and every primary in the expression is a *globally static primary*, where a globally static primary is a primary that, if it denotes an object or a function, does not denote a dynamically elaborated named entity (see 12.5) and is one of the following:

a. A literal of type TIME
b. A locally static primary
c. A generic constant
d. A generate parameter
e. A constant (including a deferred constant)
f. An alias whose aliased name (given in the corresponding alias declaration) is a globally static primary
g. An array aggregate, if and only if

    1) All expressions in its element associations are globally static expressions, and

    2) All ranges in its element associations are globally static ranges

h. A record aggregate, if and only if all expressions in its element associations are globally static expressions
i. A function call whose function name denotes a pure function and whose actual parameters are each globally static expressions
j. A predefined attribute that is a value and whose prefix is either a globally static subtype or is an object or function call that is of a globally static subtype
k. A predefined attribute that is a function, other than the predefined attributes 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING,and 'DRIVING_VALUE, whose prefix is either a globally static subtype or is an object or function call that is of a globally static subtype, and whose actual parameter (if any) is a globally static expression
l. A user-defined attribute whose value is defined by a globally static expression
m. A qualified expression whose operand is a globally static expression
n. A type conversion whose expression is a globally static expression
o. An allocator of the first form (see 7.3.6) whose subtype indication denotes a globally static subtype
p. An allocator of the second form whose qualified expression is a globally static expression
q. A globally static expression enclosed in parentheses
r. A subelement or a slice of a globally static primary, provided that any index expressions are globally static expressions and any discrete ranges used in slice names are globally static discrete ranges

A globally static range is either a range of the second form (see 3.1) whose bounds are globally static expressions, or a range of the first form whose prefix denotes either a globally static subtype or an object that is of a globally static subtype. A globally static range constraint is a range constraint whose range is globally static. A globally static scalar subtype is either a scalar base type or a scalar subtype formed by imposing on a globally static subtype a globally static range constraint. A globally static discrete range is either a globally static subtype or a globally static range.

A globally static index constraint is an index constraint for which each index subtype of the corresponding array type is globally static and in which each discrete range is globally static. A globally static array subtype is a constrained array subtype formed by imposing on an unconstrained array type a globally static index constraint. A globally static record subtype is a record type whose fields are all of globally static subtypes. A globally static access subtype is a subtype denoting an access type. A globally static file subtype is a subtype denoting a file type.

A globally static subtype is either a globally static scalar subtype, a globally static array subtype, a globally static record subtype, a globally static access subtype, or a globally static file subtype.

NOTES

1--An expression that is required to be a static expression may either be a locally static expression or a globally static expression. Similarly, a range,a range constraint, a scalar subtype, a discrete range, an index constraint, or an array subtype that is required to be static may either be locally static or globally static.

2--The rules for locally and globally static expressions imply that a declared constant or a generic may be initialized with an expression that is neither globally nor locally static; for example, with a call to an impure function. The resulting constant value may be globally or locally static, even though its subtype or its initial value expression is neither. Only interface constant, variable, and signal declarations require that their initial value expressions be static expressions.

7.5 Universal expressions

A *universal_expression* is either an expression that delivers a result of type *universal_integer* or one that delivers a result of type *universal_real*.

The same operations are predefined for the type universal_integer as for any integer type. The same operations are predefined for the type universal_real as for any floating point type. In addition, these operations include the following multiplication and division operators:

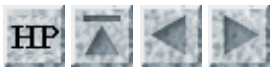
Operator	Operation	Left operand type	Right operand type	Result type
*	Mutiplication	*Universal real*	*Universal integer*	*Universal real*
		Universal real	*Universal real*	*Universal real*
/	Division	*Universal real*	*Universal integer*	*Universal real*

The accuracy of the evaluation of a universal expression of type *universal_real* is at least as good as the accuracy of evaluation of expressions of the most precise predefined floating point type supported by the implementation, apart from *universal_real* itself.

For the evaluation of an operation of a universal expression, the following rules apply. If the result is of type *universal_integer*, then the values of the operands and the result must lie within the range of the integer type with the widest range provided by the implementation, excluding type *universal_integer* itself. If the result is of type *universal_real*, then the values of the operands and the result must lie within the range of the floating point type with the widest range provided by the implementation, excluding type *universal_real* itself.

NOTE--The predefined operators for the universal types are declared in package STANDARD as shown in 14.2 .

Section 4



Declarations

The language defines several kinds of entities that are declared explicitly or implicitly by declarations.

```
declaration ::=
      type_declaration
    | subtype_declaration
    | object_declaration
    | interface_declaration
    | alias_declaration
    | attribute_declaration
    | component_declaration
    | group_template_declaration
    | group_declaration
    | entity_declaration
    | configuration_declaration
    | subprogram_declaration
    | package_declaration
```

For each form of declaration, the language rules define a certain region of text called the *scope* of the declaration (see 10.2). Each form of declaration associates an identifier with a named entity. Only within its scope, there are places where it is possible to use the identifier to refer to the associated declared entity; these places are defined by the visibility rules (see 10.3). At such places the identifier is said to be a *name* of the entity; the name is said to *denote* the associated entity.

This section describes type and subtype declarations, the various kinds of object declarations, alias declarations, attribute declarations, component declarations, and group and group template declarations. The other kinds of declarations are described in Section 1 and Section 2.

A declaration takes effect through the process of elaboration. Elaboration of declarations is discussed in Section 12.

4.1 Type declarations

A type declaration declares a type.

```
type_declaration ::=
     full_type_declaration
   | incomplete_type_declaration

full_type_declaration ::=
     type identifier is type_definition ;

type_definition ::=
     scalar_type_definition
   | composite_type_definition
   | access_type_definition
   | file_type_definition
```

The types created by the elaboration of distinct type definitions are distinct types. The elaboration of the type definition for a scalar type or a constrained array type creates both a base type and a subtype of the base type.

The simple name declared by a type declaration denotes the declared type,unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype and the base type is anonymous. A type is said to be *anonymous* if it has no simple name. For explanatory purposes, this standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier.

NOTES

1--Two type definitions always define two distinct types, even if they are lexically identical. Thus, the type definitions in the following two integer type declarations define distinct types:

```
type A is range 1 to 10;
type B is range 1 to 10;
```

This applies to type declarations for other classes of types as well.

2--The various forms of type definition are described in Section 3. Examples of type declarations are also given in that section.

4.2 Subtype declarations

A subtype declaration declares a subtype.

```
subtype_declaration ::=
    subtype identifier is subtype_indication ;

subtype_indication ::=
    [ resolution_function_name ] type_mark [ constraint ]

type_mark ::=
      type_name
    | subtype_name

constraint ::=
      range_constraint
    | index_constraint
```

A type mark denotes a type or a subtype. If a type mark is the name of a type,the type mark denotes this type and also the corresponding unconstrained subtype. The base type of a type mark is, by definition, the base type of the type or subtype denoted by the type mark.

A subtype indication defines a subtype of the base type of the type mark.

If a subtype indication includes a resolution function name, then any signal declared to be of that subtype will be resolved, if necessary, by the named function (see 2.4); for an overloaded function name, the meaning of the function name is determined by context (see 2.3 and 10.5). It is an error if the function does not meet the requirements of a resolution function (see 2.4).The presence of a resolution function name has no effect on the declarations of objects other than signals or on the declarations of files, aliases,attributes, or other subtypes.

If the subtype indication does not include a constraint, the subtype is the same as that denoted by the type mark. The condition imposed by a constraint is the condition obtained after evaluation of the expressions and ranges forming the constraint. The rules defining compatibility are given for each form of constraint in the appropriate section. These rules are such that if a constraint is compatible with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the

subtype on its values. An error occurs if any check of compatibility fails.

The direction of a discrete subtype indication is the same as the direction of the range constraint that appears as the constraint of the subtype indication. If no constraint is present, and the type mark denotes a subtype, the direction of the subtype indication is the same as that of the denoted subtype. If no constraint is present, and the type mark denotes a type, the direction of the subtype indication is the same as that of the range used to define the denoted type. The direction of a discrete subtype is the same as the direction of its subtype indication.

A subtype indication denoting an access type or a file type may not contain are solution function. Furthermore, the only allowable constraint on a subtype indication denoting an access type is an index constraint (and then only if the designated type is an array type).

NOTE--A subtype declaration does not define a new type.

4.3 Objects

An *object* is a named entity that contains (has) a value of a given type. An object is one of the following:

-- An object declared by an object declaration (see 4.3.1)

-- A loop or generate parameter (see 8.9 and 9.7)

-- A formal parameter of a subprogram (see 2.1.1)

-- A formal port (see 1.1.1.2 and 9.1)

-- A formal generic (see 1.1.1.1 and 9.1)

-- A local port (see 4.5)

-- A local generic (see 4.5)

-- An implicit signal GUARD defined by the guard expression of a block statement (see 9.1)

In addition, the following are objects, but are not named entities:

-- An implicit signal defined by any of the predefined attributes 'DELAYED,'STABLE, 'QUIET, and 'TRANSACTION (see 14.1)

-- An element or slice of another object (see 6.3 , 6.4 , and 6.5)

-- An object designated by a value of an access type (see 3.3)

There are four classes of objects: constants, signals, variables, and files. The variable class of objects also has an additional subclass: shared variables. The class of an explicitly declared object is specified by the reserved word that must or may appear at the beginning of the declaration of that object. For a given object of a composite type, each subelement of that object is itself an object of the same class and subclass, if any, as the given object. The value of a composite object is the aggregation of the values of its subelements.

Objects declared by object declarations are available for use within blocks,processes, subprograms, or packages. Loop and generate parameters are implicitly declared by the corresponding statement and are available for use only within that statement. Other objects, declared by interface declarations,create channels for the communication of values between independent parts of

a description.

4.3.1 Object declarations

An object declaration declares an object of a specified type. Such an object is called an *explicitly declared object*.

```
object_declaration ::=
      constant_declaration
    | signal_declaration
    | variable_declaration
    | file_declaration
```

An object declaration is called a *single-object declaration* if its identifier list has a single identifier; it is called a *multiple-object declaration* if the identifier list has two or more identifiers. A multiple-object declaration is equivalent to a sequence of the corresponding number of single-object declarations. For each identifier of the list, the equivalent sequence has a single-object declaration formed by this identifier,followed by a colon and by whatever appears at the right of the colon in the multiple-object declaration; the equivalent sequence is in the same order as the identifier list.

A similar equivalence applies also for interface object declarations (see 4.3.2).

NOTE--The subelements of a composite, declared object are not declared objects.

4.3.1.1 Constant declarations

A constant declaration declares a *constant* of the specified type. Such a constant is an *explicitly declared constant*.

```
constant_declaration ::=
    constant identifier_list : subtype_indication [ := expression ] ;
```

If the assignment symbol ":=" followed by an expression is present in a constant declaration, the expression specifies the value of the constant; the type of the expression must be that of the constant. The value of a constant cannot be modified after the declaration is elaborated.

If the assignment symbol ":=" followed by an expression is not present in a constant declaration, then the declaration declares a *deferred constant*. Such a constant declaration may only appear in a package declaration. The corresponding full constant declaration, which defines the value of the constant, must appear in the body of the package (see 2.6).

Formal parameters of subprograms that are of mode **in** may be constants,and local and formal generics are always constants; the declarations of such objects are discussed in 4.3.2 . A loop parameter is a constant within the corresponding loop (see 8.9); similarly, a generate parameter is a constant within the corresponding generate statement (see 9.7). A subelement or slice of a constant is a constant.

It is an error if a constant declaration declares a constant that is of a file type, an access type, or a composite type that has a subelement that is a file type or an access type.

NOTE--The subelements of a composite, declared constant are not declared constants.

Examples:

```
constant TOLERANCE : DISTANCE := 1.5 nm;
constant PI : REAL := 3.141592;
constant CYCLE_TIME : TIME := 100 ns;
constant Propagation_Delay : DELAY_LENGTH;   -- a deferred constant
```

4.3.1.2 Signal declarations

A signal declaration declares a *signal* of the specified type. Such a signal is an *explicitly declared signal*.

```
signal_declaration ::=
    signal identifier_list : subtype_indication [ signal_kind ] [ := expression
] ;

signal_kind ::=  register  |  bus
```

If the name of a resolution function appears in the declaration of a signal or in the declaration of the subtype used to declare the signal, then that resolution function is associated with the declared signal. Such a signal is called a *resolved signal*.

If a signal kind appears in a signal declaration, then the signals so declared are *guarded* signals of the kind indicated. For a guarded signal that is of a composite type, each subelement is likewise a guarded signal. For a guarded signal that is of an array type, each slice (see 6.5) is likewise a guarded signal. A guarded signal may be assigned values under the control of Boolean-valued *guard expressions* (or *guards*).

When a given guard becomes False, the drivers of the corresponding guarded signals are implicitly assigned a null transaction (see 8.4.1) to cause those drivers to turn off. A disconnection specification (see 5.3) is used to specify the time required for those drivers to turn off.

If the signal declaration includes the assignment symbol followed by an expression, it must be of the same type as the signal. Such an expression is said to be a *default expression*. The default expression defines a *default value* associated with the signal or, for a composite signal,with each scalar subelement thereof. For a signal declared to be of a scalar subtype, the value of the default expression is the default value of the signal. For a signal declared to be of a composite subtype, each scalar subelement of the value of the default expression is the default value of the corresponding subelement of the signal.

In the absence of an explicit default expression, an implicit default value is assumed for a signal of a scalar subtype or for each scalar subelement of a composite signal, each of which is itself a signal of a scalar subtype. The implicit default value for a signal of a scalar subtype T is defined to be that given by T'LEFT.

It is an error if a signal declaration declares a signal that is of a file type or an access type. It is also an error if a guarded signal of a scalar type is neither a resolved signal nor a subelement of a resolved signal.

A signal may have one or more *sources*. For a signal of a scalar type,each source is either a driver (see 12.6.1) or an **out**, **inout**,**buffer**, or **linkage** port of a component instance or of a block statement with which the signal is associated. For a signal of a composite type, each composite source is a collection of scalar sources, one for each scalar subelement of the signal. It is an error if, after the elaboration of a description, a signal has multiple sources and it is not a resolved signal. It is also an error if, after the elaboration of a description, a resolved signal has more sources than the number of elements in the index range of the type of the formal parameter of the resolution function associated with the resolved signal.

If a subelement or slice of a resolved signal of composite type is associated as an actual in a port map aspect (either in a component instantiation statement or in a binding indication), and if the corresponding formal is of mode **out**, **inout**, **buffer**, or **linkage**, then every scalar subelement of that signal must be associated exactly once with such a formal in the same port map aspect, and the collection of the corresponding formal parts taken together constitute one source of the signal. If a resolved signal of composite type is associated as an actual in a port map aspect, that is equivalent to each of its subelements being associated in the same port map aspect.

If a subelement of a resolved signal of composite type has a driver in a given process, then every scalar subelement of that signal must have a driver in the same process, and the collection of all of those drivers taken together constitute one source of the signal.

The default value associated with a scalar signal defines the value component of a transaction that is the initial contents of each

driver (if any) of that signal. The time component of the transaction is not defined, but the transaction is understood to have already occurred by the start of simulation.

Examples:

```
signal S : STANDARD.BIT_VECTOR (1 to 10) ;

signal CLK1, CLK2 : TIME ;

signal OUTPUT : WIRED_OR MULTI_VALUED_LOGIC;
```

NOTES

1--Ports of any mode are also signals. The term *signal* is used in this standard to refer to objects declared either by signal declarations or by port declarations (or to subelements, slices, or aliases of such objects). It also refers to the implicit signal GUARD (see 9.1) and to implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, and 'TRANSACTION. The term *port* is used to refer to objects declared by port declarations only.

2--Signals are given initial values by initializing their drivers. The initial values of drivers are then propagated through the corresponding net to determine the initial values of the signals that make up the net (see 12.6.3).

3--The value of a signal may be indirectly modified by a signal assignment statement (see 8.4); such assignments affect the future values of the signal.

4--The subelements of a composite, declared signal are not declared signals.

Cross-References: Disconnection specifications, 5.3 ; Disconnection statements, 9.5 ; Guarded assignment, 9.5 ; Guarded blocks, 9.1 ; Guarded targets, 9.5 ; Signal guard, 9.1 .

4.3.1.3 Variable declarations

A variable declaration declares a *variable* of the specified type. Such a variable is an *explicitly declared variable*.

```
variable_declaration ::=
    [ shared ] variable identifier_list : subtype_indication [ := expression ] ;
```

A variable declaration that includes the reserved word **shared** is a *shared variable declaration*. A shared variable declaration declares a *shared variable*. Shared variables are a subclass of the variable class of objects. More than one process may access a given shared variable; however,if more than one process accesses a given shared variable during the same simulation cycle (see 12.6.4), neither the value of the shared variable after the access nor the value read from the shared variable is defined by the language. A description is erroneous if it depends on whether or how an implementation sequentializes access to shared variables.

If the variable declaration includes the assignment symbol followed by an expression, the expression specifies an initial value for the declared variable; the type of the expression must be that of the variable. Such an expression is said to be an *initial value expression*.

If an initial value expression appears in the declaration of a variable, then the initial value of the variable is determined by that expression each time the variable declaration is elaborated. In the absence of an initial value expression, a default initial value applies. The default initial value for a /variable of a scalar subtype T is defined to be the value given by T'LEFT. The default initial value of a variable of a composite type is defined to be the aggregate of the default initial values of all of its scalar subelements, each of which is itself a variable of a scalar subtype. The default initial value of a variable of an access type is defined to be the value **null** for that type.

NOTES

1--The value of a variable may be modified by a variable assignment statement (see 8.5); such assignments take effect immediately.

2--The variables declared within a given procedure persist until that procedure completes and returns to the caller. For procedures that contain wait statements, a variable may therefore persist from one point in simulation time to another, and the value in the variable is thus maintained over time. For processes, which never complete, all variables persist from the beginning of simulation until the end of simulation.

3--The subelements of a composite, declared variable are not declared variables.

4--Since the language does not guarantee the synchronization of accesses to shared variables by multiple processes in the same simulation cycle, the use of shared variables in this manner is non portable and nondeterministic. For example, consider the following architecture:

```
architecture UseSharedVariables of SomeEntity is
    subtype ShortRange is INTEGER range 0 to 1;
    shared variable Counter: ShortRange := 0;

begin
PROC1: process
   begin
       Counter := Counter + 1;     -- The subtype check may or may not fail.
       wait;
   end process PROC1;

 PROC2: process
   begin
       Counter := Counter - 1;     -- The subtype check may or may not fail.
       wait;
   end process PROC2;
end architecture UseSharedVariables;
```

In particular, the value of Counter after the execution of both processes is not guaranteed to be either 0 or 1, even if Counter is declared to be of type INTEGER.

5--Variables declared immediately within entity declarations, architecture bodies, packages, package bodies, and blocks must be shared variables. Variables declared immediately within subprograms and processes must not be shared variables.

Examples:

```
variable INDEX : INTEGER range 0 to 99 := 0 ;
    -- Initial value is determined by the initial value expression

variable COUNT : POSITIVE ;
    -- Initial value is POSITIVE'LEFT; that is,1.

variable MEMORY : BIT_MATRIX (0 to 7, 0 to 1023) ;
    -- Initial value is the aggregate of the initial values of each element
```

4.3.1.4 File declarations

A file declaration declares a *file* of the specified type. Such a file is an *explicitly declared file*.

```
     file_declaration ::=
          file identifier_list : subtype_indication [ file_open_information ] ;

     file_open_information ::=  [ open file_open_kind_expression ] is
file_logical_name

     file_logical_name ::=  string_expression
```

The subtype indication of a file declaration must define a file subtype.

If file open information is included in a given file declaration, then the file declared by the declaration is opened (see 3.4.1) with an implicit call to FILE_OPEN when the file declaration is elaborated (see 12.3.1.4). This implicit call is to the FILE_OPEN procedure of the first form, and it associates the identifier with the file parameter F, the file logical name with the External_Name parameter, and the file open kind expression with the Open_Kind parameter. If a file open kind expression is not included in the file open information of a given file declaration, then the default value of READ_MODE is used during elaboration of the file declaration.

If file open information is not included in a given file declaration, then the file declared by the declaration is not opened when the file declaration is elaborated.

The file logical name must be an expression of predefined type STRING. The value of this expression is interpreted as a logical name for a file in the host system environment. An implementation must provide some mechanism to associate a file logical name with a host-dependent file. Such a mechanism is not defined by the language.

The file logical name identifies an external file in the host file system that is associated with the file object. This association provides a mechanism for either importing data contained in an external file into the design during simulation or exporting data generated during simulation to an external file.

If multiple file objects are associated with the same external file, and each file object has an access mode that is read-only (see 3.4.1), then values read from each file object are read from the external file associated with the file object. The language does not define the order in which such values are read from the external file, nor does it define whether each value is read once or multiple times (once per file object).

The language does not define the order of and the relationship, if any, between values read from and written to multiple file objects that are associated with the same external file. An implementation may restrict the number of file objects that may be associated at one time with a given external file.

If a formal subprogram parameter is of the class **file**, it must be associated with an actual that is a file object.

Examples:

```
     type IntegerFile is file of INTEGER;

     file F1: IntegerFile;                  --  No implicit FILE_OPEN is performed
                                            --  during elaboration.

     file F2: IntegerFile is "test.dat";    --  At elaboration, an implicit call is
performed:
                                            --  FILE_OPEN (F2, "test.dat");
                                            --  The OPEN_KIND parameter defaults to
                                            --  READ_MODE.

     file F3: IntegerFile open WRITE_MODE is "test.dat";
                                            --  At elaboration, an implicit call is
performed:
                                            --  FILE_OPEN (F3, "test.dat",
```

```
WRITE_MODE);
```

NOTE--All file objects associated with the same external file should be of the same base type.

4.3.2 Interface declarations

An interface declaration declares an *interface object* of a specified type. Interface objects include *interface constants* that appear as generics of a design entity, a component, or a block, or as constant parameters of subprograms; *interface signals* that appear as ports of a design entity, component, or block, or as signal parameters of subprograms;*interface variables* that appear as variable parameters of subprograms; and *interface files* that appear as file parameters of subprograms.

```
     interface_declaration ::=
            interface_constant_declaration
          | interface_signal_declaration
          | interface_variable_declaration
          | interface_file_declaration

     interface_constant_declaration ::=
          [constant] identifier_list : [ in ] subtype_indication [ :=
static_expression ]

     interface_signal_declaration ::=
          [signal] identifier_list : [ mode ] subtype_indication [ bus ] [ :=
static_expression ]

     interface_variable_declaration ::=
          [variable] identifier_list : [ mode ] subtype_indication [ :=
static_expression ]

     interface_file_declaration ::=
          file identifier_list subtype_indication

     mode ::=  in | out | inout | buffer | linkage
```

If no mode is explicitly given in an interface declaration other than an interface file declaration, mode **in** is assumed.

For an interface constant declaration or an interface signal declaration, the subtype indication must define a subtype that is neither a file type nor an access type.

For an interface file declaration, it is an error if the subtype indication does not denote a subtype of a file type.

If an interface signal declaration includes the reserved word **bus**, then the signal declared by that interface declaration is a guarded signal of signal kind **bus**.

If an interface declaration contains a ":=" symbol followed by an expression,the expression is said to be the *default expression* of the interface object. The type of a default expression must be that of the corresponding interface object. It is an error if a default expression appears in an interface declaration and any of the following conditions hold:

-- The mode is **linkage**

-- The interface object is a formal signal parameter

-- The interface object is a formal variable parameter of mode other than **in**.

In an interface signal declaration appearing in a port list, the default expression defines the default value(s) associated with the

interface signal or its subelements. In the absence of a default expression, an implicit default value is assumed for the signal or for each scalar subelement, as defined for signal declarations (see 4.3.1.2). The value, whether implicitly or explicitly provided, is used to determine the initial contents of drivers, if any, of the interface signal as specified for signal declarations.

An interface object provides a channel of communication between the environment and a particular portion of a description. The value of an interface object may be determined by the value of an associated object or expression in the environment; similarly, the value of an object in the environment may be determined by the value of an associated interface object. The manner in which such associations are made is described in 4.3.2.2 .

The value of an object is said to be *read* when one of the following conditions is satisfied:

-- When the object is evaluated, and also (indirectly) when the object is associated with an interface object of the modes **in**, **inout**, or **linkage**.

-- When the object is a signal and a name denoting the object appears in a sensitivity list in a wait statement or a process statement.

-- When the object is a signal and the value of any of its predefined attributes 'STABLE, 'QUIET, 'DELAYED, 'TRANSACTION, 'EVENT, 'ACTIVE,'LAST_EVENT, 'LAST_ACTIVE, or 'LAST_VALUE is read.

-- When one of its subelements is read.

-- When the object is a file and a READ operation is performed on the file.

The value of an object is said to be *updated* when one of the following conditions is satisfied:

-- When it is the target of an assignment, and also (indirectly) when the object is associated with an interface object of the modes **out**,**buffer**, **inout**, or **linkage**.

-- When one of its subelements is updated.

-- When the object is a file and a WRITE operation is performed on the file.

Only signal, variable, or file objects may be updated.

An interface object has one of the following modes:

-- **in.** The value of the interface object may only be read. In addition, any attributes of the interface object may be read, except that attributes 'STABLE, 'QUIET, 'DELAYED, and 'TRANSACTION of a subprogram signal parameter may not be read within the corresponding subprogram. For a file object, operation ENDFILE is allowed.

-- **out.** The value of the interface object may be updated. Reading the attributes of the interface element, other than the predefined attributes 'STABLE, 'QUIET, 'DELAYED, 'TRANSACTION, 'EVENT, 'ACTIVE, 'LAST_EVENT,'LAST_ACTIVE, and 'LAST_VALUE, is allowed. No other reading is allowed.

-- **inout.** The value of the interface object may be both read and updated. Reading the attributes of the interface object, other than the attributes 'STABLE, 'QUIET, 'DELAYED, and 'TRANSACTION of a signal parameter,is also permitted. For a file object, all file operations (see 3.4.1) are allowed.

-- **buffer.** The value of the interface object may be both read and updated. Reading the attributes of the interface object is also permitted.

-- **linkage.** The value of the interface object may be read or updated,but only by appearing as an actual corresponding to an interface object of mode **linkage**. No other reading or updating is permitted.

NOTES

1--Although signals of modes **inout** and **buffer** have the same characteristics with respect to whether they may be read or updated, a signal of mode **inout** may be updated by zero or more sources, whereas a signal of mode **buffer** must be updated by at most one source (see 1.1.1.2).

2--A subprogram parameter that is of a file type must be declared as a file parameter.

3--Since shared variables are a subclass of variables, a shared variable may be associated as an actual with a formal of class variable.

4.3.2.1 Interface lists

An interface list contains the declarations of the interface objects required by a subprogram, a component, a design entity, or a block statement.

```
interface_list ::=
     interface_element { ; interface_element }

interface_element ::=  interface_declaration
```

A *generic* interface list consists entirely of interface constant declarations. A *port* interface list consists entirely of interface signal declarations. A *parameter* interface list may contain interface constant declarations, interface signal declarations, interface variable declarations, interface file declarations, or any combination thereof.

A name that denotes an interface object may not appear in any interface declaration within the interface list containing the denoted interface object except to declare this object.

NOTE--The above restriction makes the following three interface lists illegal:

```
     entity E is
          generic     (G1:   INTEGER;     G2:   INTEGER := G1);                    --
illegal
          port        (P1:   STRING;      P2:   STRING(P1'RANGE));
-- illegal

          procedure X (Y1, Y2: INTEGER;   Y3:   INTEGER range Y1 to Y2);
-- illegal
     end E;
```

However,the following interface lists are legal:

```
     entity E is
          generic      (G1, G2, G3, G4:    INTEGER);
          port         (P1, P2:            STRING (G1 to G2));

          procedure X  (Y3:                INTEGER range G3 to G4);
     end E;
```

4.3.2.2 Association lists

An association list establishes correspondences between formal or local generic, port, or parameter names on the one hand and local or actual names or expressions on the other.

```
association_list ::=
    association_element { , association_element }

association_element ::=
    [ formal_part => ] actual_part

formal_part ::=
     formal_designator
    | function_name ( formal_designator )
    | type_mark ( formal_designator )

formal_designator ::=
     generic_name
    | port_name
    | parameter_name

actual_part ::=
      actual_designator
    | function_name ( actual_designator )
    | type_mark ( actual_designator )

actual_designator ::=
      expression
    | signal_name
    | variable_name
    | file_name
    | open
```

Each association element in an association list associates one actual designator with the corresponding interface element in the interface list of a subprogram declaration, component declaration, entity declaration, or block statement. The corresponding interface element is determined either by position or by name.

An association element is said to be *named* if the formal designator appears explicitly; otherwise, it is said to be *positional*. For a positional association, an actual designator at a given position in an association list corresponds to the interface element at the same position in the interface list.

Named associations can be given in any order, but if both positional and named associations appear in the same association list, then all positional associations must occur first at their normal position. Hence once a named association is used, the rest of the association list must use only named associations.

In the following, the term *actual* refers to an actual designator, and the term *formal* refers to a formal designator.

The formal part of a named element association may be in the form of a function call, where the single argument of the function is the formal designator itself, if and only if the mode of the formal is **out**, **inout**,**buffer**, or **linkage**, and if the actual is not **open**. In this case, the function name must denote a function whose single parameter is of the type of the formal and whose result is the type of the corresponding actual. Such a *conversion function* provides for type conversion in the event that data flows from the formal to the actual.

Alternatively, the formal part of a named element association may be in the form of a type conversion, where the expression to be converted is the formal designator itself, if and only if the mode of the formal is **out**,**inout**, **buffer**, or **linkage**, and if the actual is not **open**. In this case, the base type denoted by the type mark must be the same as the base type of the corresponding actual. Such a type conversion provides for type conversion in the event that data flows from the formal to the actual. It is an error if the type of the formal is not closely related to the type of the actual. (See 7.3.5 .)

Similarly, the actual part of a (named or positional) element association maybe in the form of a function call, where the single argument of the function is the actual designator itself, if and only if the mode of the formal is **in**, **inout**, or **linkage**, and if the

actual is not **open**. In this case, the function name must denote a function whose single parameter is of the type of the actual, and whose result is the type of the corresponding formal. In addition, the formal must not be of class **constant** for this interpretation to hold (the actual is interpreted as an expression that is a function call if the class of the formal is **constant**). Such a conversion function provides for type conversion in the event that data flows from the actual to the formal.

Alternatively, the actual part of a (named or positional) element association may be in the form of a type conversion, where the expression to be type converted is the actual designator itself, if and only if the mode of the formal is **in**, **inout**, or **linkage**, and if the actual is not **open**. In this case, the base type denoted by the type mark must be the same as the base type of the corresponding formal. Such a type conversion provides for type conversion in the event that data flows from the actual to the formal. It is an error if the type of the actual is not closely related to the type of the formal.

The type of the actual (after applying the conversion function or type conversion, if present in the actual part) must be the same as the type of the corresponding formal, if the mode of the formal is **in**, **inout**, or **linkage**, and if the actual is not **open**. Similarly, if the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and if the actual is not **open**, then the type of the formal (after applying the conversion function or type conversion, if present in the formal part) must be the same as the corresponding actual.

For the association of signals with corresponding formal ports, association of a formal of a given composite type with an actual of the same type is equivalent to the association of each scalar subelement of the formal with the matching subelement of the actual, provided that no conversion function or type conversion is present in either the actual part or the formal part of the association element. If a conversion function or type conversion is present,then the entire formal is considered to be associated with the entire actual.

Similarly, for the association of actuals with corresponding formal subprogram parameters, association of a formal parameter of a given composite type with an actual of the same type is equivalent to the association of each scalar subelement of the formal parameter with the matching subelement of the actual. Different parameter passing mechanisms may be required in each case, but in both cases the associations will have an equivalent effect. This equivalence applies provided that no actual is accessible by more than one path (see 2.1.1.1).

A formal may be either an explicitly declared interface object or member (see Section 3) of such an interface object. In the former case,such a formal is said to be *associated in whole*. In the latter cases,named association must be used to associate the formal and actual; the subelements of such a formal are said to be *associated individually*. Furthermore, every scalar subelement of the explicitly declared interface object must be associated exactly once with an actual (or subelement thereof)in the same association list, and all such associations must appear in a contiguous sequence within that association list. Each association element that associates a slice or subelement (or slice thereof) of an interface object must identify the formal with a locally static name.

If an interface element in an interface list includes a default expression fora formal generic, for a formal port of any mode other than **linkage**, or for a formal variable or constant parameter of mode **in**, then any corresponding association list need not include an association element for that interface element. If the association element is not included in the association list, or if the actual is **open**, then the value of the default expression is used as the actual expression or signal value in an implicit association element for that interface element.

It is an error if an actual of **open** is associated with a formal that is associated individually. An actual of **open** counts as the single association allowed for the corresponding formal but does not supply a constant, signal, or variable (as is appropriate to the object class of the formal) to the formal.

NOTES

1--It is a consequence of these rules that, if an association element is omitted from an association list in order to make use of the default expression on the corresponding interface element, all subsequent association elements in that association list must be named associations.

2--Although a default expression can appear in an interface element that declares a (local or formal) port, such a default expression is not interpreted as the value of an implicit association element for that port. Instead, the value of the expression is

used to determine the effective value of that port during simulation if the port is left unconnected (see 12.6.2).

3--Named association may not be used when invoking implicitly defined operations, since the formal parameters of these operators are not named (see 7.2).

4--Since information flows only from the actual to the formal when the mode of the formal is **in**, and since a function call is itself an expression, the actual associated with a formal of object class **constant** is never interpreted as a conversion function or a type conversion converting an actual designator that is an expression. Thus, the following association element is legal:

```
Param => F (open)
```

under the conditions that Param is a constant formal and F is a function returning the same base type as that of Param and having one or more parameters, all of which may be defaulted.

5--No conversion function or type conversion may appear in the actual part when the actual designator is **open**.

4.3.3 Alias declarations

An alias declaration declares an alternate name for an existing named entity.

```
alias_declaration ::=
     alias alias_designator [ : subtype_indication ] is name [ signature ] ;

alias_designator ::=  identifier | character_literal | operator_symbol
```

An *object alias* is an alias whose alias designator denotes an object (that is, a constant, a variable, a signal, or a file). A *nonobject alias* is an alias whose alias designator denotes some named entity other than an object. An alias can be declared for all named entities except for labels, loop parameters, and generate parameters.

The alias designator in an alias declaration denotes the named entity specified by the name and, if present, the signature in the alias declaration. An alias of a signal denotes a signal; an alias of a variable denotes a variable; an alias of a constant denotes a constant; and an alias of a file denotes a file. Similarly, an alias of a subprogram (including an operator) denotes a subprogram, an alias of an enumeration literal denotes an enumeration literal,and so forth.

NOTES

1--Since, for example, the alias of a variable is a variable, every reference within this document to a designator (a name, character literal, or operator symbol) that requires the designator to denote a named entity with certain characteristics (for example, to be a variable) allows the designator to denote an alias, so long as the aliased name denotes a named entity having the required characteristics. This situation holds except where aliases are specifically prohibited.

2--The alias of an overloadable object is itself overloadable.

4.3.3.1 Object aliases

The following rules apply to object aliases:

a. A signature may not appear in a declaration of an object alias.
b. The name must be a static name (see 6.1) that denotes an object. The base type of the name specified in an alias declaration must be the same as the base type of the type mark in the subtype indication (if the subtype indication is present); this type must not be a multi-dimensional array type. When the object denoted by the name is referenced via the alias defined by the alias declaration, the following rules apply:

   -- If the subtype indication is absent or if it is present and denotes an unconstrained array type:

-- If the alias designator denotes a slice of an object, then the subtype of the object is viewed as if it were of the subtype specified by the slice

-- Otherwise, the object is viewed as if it were of the subtype specified in the declaration of the object denoted by the name

-- If the subtype indication is present and denotes a constrained array subtype, then the object is viewed as if it were of the subtype specified by the subtype indication; moreover, the subtype denoted by the subtype indication must include a matching element (see 7.2.2) for each element of the object denoted by the name;

-- If the subtype indication denotes a scalar subtype, then the object is viewed as if it were of the subtype specified by the subtype indication;moreover, it is an error if this subtype does not have the same bounds and direction as the subtype denoted by the object name.

a. The same applies to attribute references where the prefix of the attribute name denotes the alias.

b. A reference to an element of an object alias is implicitly a reference to the matching element of the object denoted by the alias. A reference to a slice of an object alias consisting of the elements e_1, e_2, ..., e_n is implicitly a reference to a slice of the object denoted by the alias consisting of the matching elements corresponding to each of e_1 through e_n.

4.3.3.2 Nonobject aliases

The following rules apply to nonobject aliases:

a. A subtype indication may not appear in a nonobject alias.

b. A signature is required if the name denotes a subprogram (including an operator) or enumeration literal. In this case, the signature is required to match (see 2.3) the parameter and result type profile of exactly one of the subprograms or enumeration literals denoted by the name.

c. If the name denotes an enumeration type, then one implicit alias declaration for each of the literals of the type immediately follows the alias declaration for the enumeration type; each such implicit declaration has, as its alias designator, the simple name or character literal of the literal and has, as its name, a name constructed by taking the name of the alias for thee numeration type and substituting the simple name or character literal being aliased for the simple name of the type. Each implicit alias has a signature that matches the parameter and result type profile of the literal being aliased.

d. Alternatively, if the name denotes a physical type, then one implicitali as declaration for each of the units of the type immediately follows the alias declaration for the physical type; each such implicit declaration has, as its name, a name constructed by taking the name of the alias for the physical type and substituting the simple name of the unit being aliased for the simple name of the type.

e. Finally, if the name denotes a type, then implicit alias declarations for each predefined operator for the type immediately follow the explicit alias declaration for the type and, if present, any implicit alias declarations for literals or units of the type. Each implicit alias has a signature that matches the parameter and result type profile of the implicit operator being aliased.

Examples:

```
variable REAL_NUMBER : BIT_VECTOR (0 to 31);

alias SIGN : BIT is REAL_NUMBER (0);
    -- SIGN is now a scalar (BIT) value
```

```
alias MANTISSA : BIT_VECTOR (23 downto 0) is REAL_NUMBER (8 to 31);
    -- MANTISSA is a 24b value whose range is 23 downto 0.
    -- Note that the ranges of MANTISSA and REAL_NUMBER (8 to 31)
    -- have opposite directions.  A reference to MANTISSA (23 downto 18)
    -- is equivalent to a reference to REAL_NUMBER (8 to 13).

alias EXPONENT : BIT_VECTOR (1 to 7) is REAL_NUMBER (1 to 7);
     -- EXPONENT is a 7-bit value whose range is 1 to 7.

alias STD_BIT   is STD.STANDARD.BIT;                --   explicit alias

--  alias '0'   is STD.STANDARD.'0'        --   implicit aliases ...
--                          [return STD.STANDARD.BIT];
-- alias '1'   is STD.STANDARD.'1'
--                          [return STD.STANDARD.BIT];
-- alias "and" is STD.STANDARD."and"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BIT];
-- alias "or"  is STD.STANDARD."or"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BIT];
-- alias "nand" is STD.STANDARD."nand"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BIT];
-- alias "nor"  is STD.STANDARD."nor"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BIT];
-- alias "xor" is STD.STANDARD."xor"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BIT];
-- alias "xnor is STD.STANDARD."xnor"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BIT];
-- alias "not" is STD.STANDARD."not"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BIT];
-- alias "="   is STD.STANDARD."="
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BOOLEAN];
-- alias "/="  is STD.STANDARD."/="
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BOOLEAN];
-- alias "<"   is STD.STANDARD."<"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BOOLEAN];
-- alias "<="  is STD.STANDARD."<="
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BOOLEAN];
-- alias ">"  is STD.STANDARD.">"
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BOOLEAN];
-- alias ">="  is STD.STANDARD.">="
--                      [STD.STANDARD.BIT, STD.STANDARD.BIT
--                          return STD.STANDARD.BOOLEAN];
```

NOTE--An alias of an explicitly declared object is not an explicitly declared object,nor is the alias of a subelement or slice of an

explicitly declared object an explicitly declared object.

4.4 Attribute declarations

An attribute is a value, function, type, range, signal, or constant that may be associated with one or more named entities in a description. There are two categories of attributes: predefined attributes and user-defined attributes. Predefined attributes provide information about named entities in a description. Section 14 contains the definition of all predefined attributes. Predefined attributes that are signals may not be updated.

User-defined attributes are constants of arbitrary type. Such attributes are defined by an attribute declaration.

```
attribute_declaration ::=
     attribute identifier: type_mark ;
```

The identifier is said to be the *designator* of the attribute. An attribute may be associated with an entity declaration, an architecture, a configuration,a procedure, a function, a package, a type, a subtype, a constant, a signal, a variable, a component, a label, a literal, a unit, a group, or a file.

The type mark must denote a subtype that is neither an access type nor a file type. The subtype need not be constrained.

Examples:

```
type COORDINATE is record X,Y: INTEGER; end record;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
attribute LOCATION: COORDINATE;
attribute PIN_NO: POSITIVE;
```

NOTES

1--A given named entity E will be decorated with the user-defined attribute A if and only if an attribute specification for the value of attribute A exists in the same declarative part as the declaration of E. In the absence of such a specification, an attribute name of the form E'A is illegal.

2--A user-defined attribute is associated with the named entity denoted by the name specified in a declaration, not with the name itself. Hence, an attribute of an object can be referenced by using an alias for that object rather than the declared name of the object as the prefix of the attribute name, and the attribute referenced in such a way is the same attribute (and therefore has the same value) as the attribute referenced by using the declared name of the object as the prefix.

3--A user-defined attribute of a port, signal, variable, or constant of some composite type is an attribute of the entire port, signal, variable, or constant, not of its elements. If it is necessary to associate an attribute with each element of some composite object, then the attribute itself can be declared to be of a composite type such that for each element of the object,there is a corresponding element of the attribute.

4.5 Component declarations

A component declaration declares a virtual design entity interface that may be used in a component instantiation statement. A component configuration or a configuration specification can be used to associate a component instance with a design entity that resides in a library.

```
component_declaration ::=
   component identifier [ is ]
      [ local_generic_clause ]
```

```
        [ local_port_clause ]
    end component [ component_simple_name ] ;
```

Each interface object in the local generic clause declares a local generic. Each interface object in the local port clause declares a local port.

If a simple name appears at the end of a component declaration, it must repeat the identifier of the component declaration.

4.6 Group template declarations

A group template declaration declares a *group template*, which defines the allowable classes of named entities that can appear in a group.

```
group_template_declaration ::=
    group identifier is ( entity_class_entry_list ) ;

entity_class_entry_list ::=
    entity_class_entry { , entity_class_entry }

entity_class_entry ::=  entity_class [ <> ]
```

A group template is characterized by the number of entity class entries and the entity class at each position. Entity classes are described in 5.1 .

An entity class entry that is an entity class defines the entity class that may appear at that position in the group type. An entity class entry that includes a box (<>) allows zero or more group constituents to appear in this position in the corresponding group declaration; such an entity class entry must be the last one within the entity class entry list.

Examples:

```
    group PIN2PIN is (signal, signal);          --  Groups of this type consist of
two signals.

    group RESOURCE is (label <>);               --  Groups of this type consist of
any number
                                                --  of labels.
    group DIFF_CYCLES is (group <>);            --  A group of groups.
```

4.7 Group declarations

A group declaration declares a *group*, a named collection of named entities. Named entities are described

in 5.1 .

```
group_declaration ::=
    group identifier : group_template_name ( group_constituent_list ) ;

group_constituent_list ::=  group_constituent { , group_constituent }

group_constituent ::=  name | character_literal
```

It is an error if the class of any group constituent in the group constituent list is not the same as the class specified by the corresponding entity class entry in the entity class entry list of the group template.

A name that is a group constituent may not be an attribute name (see 6.6), nor, if it contains a prefix, may that prefix be a function call.

If a group declaration appears within a package body, and a group constituent within that group declaration is the same as the simple name of the package body, then the group constituent denotes the package declaration and not the package body. The same rule holds for group declarations appearing within subprogram bodies containing group constituents with the same designator as that of the enclosing subprogram body.

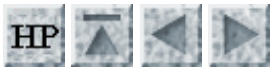
If a group declaration contains a group constituent that denotes a variable of an access type, the group declaration declares a group incorporating the variable itself, and not the designated object, if any.

Examples:

```
     group G1: RESOURCE (L1, L2);                    --  A group of two labels.

     group G2: RESOURCE (L3, L4, L5);                --  A group  of three labels.

     group C2Q: PIN2PIN (PROJECT.GLOBALS.CK, Q);     --  Groups may associate named
                                                     --  entities in different
declarative                                 --  parts (and regions).
     group CONSTRAINT1: DIFF_CYCLES (G1, G3);        --  A group of groups.
```



Section 10

Scope and visibility

The rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the text of the description are presented in this section. The formulation of these rules uses the notion of a declarative region.

10.1 Declarative region

A declarative region is a portion of the text of the description. A single declarative region is formed by the text of each of the following:

a. An entity declaration, together with a corresponding architecture body.

b. A configuration declaration.

c. A subprogram declaration, together with the corresponding subprogram body.

d. A package declaration, together with the corresponding body (if any).

e. A record type declaration.

f. A component declaration.

g. A block statement.

h. A process statement.

i. A loop statement.

j. A block configuration.

k. A component configuration.

l. A generate statement.

In each of these cases, the declarative region is said to be *associated* with the corresponding declaration or statement. A declaration is said to occur *immediately within* a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself.

Certain declarative regions include disjoint parts. Each declarative region is nevertheless considered as a (logically) continuous portion of the description text. Hence, if any rule defines a portion of text as the text that *extends* from some specific point of a declarative region to the end of this region, then this portion is the corresponding subset of the declarative region (thus, it does not include intermediate declarative items between the interface declaration and a corresponding body declaration).

10.2 Scope of declarations

For each form of declaration, the language rules define a certain portion of the description text called the *scope of the declaration*. The scope of a declaration is also called the scope of any named entity declared by the declaration. Furthermore, if the declaration associates some notation (either an identifier, a character literal, or an operator symbol) with the named entity, this portion of the text is also called the scope of this notation. Within the scope of a named entity, and only there, there are places where it is legal to use the associated notation in order to refer to the named entity. These places are defined by the rules of visibility and overloading.

The scope of a declaration that occurs immediately within a declarative region extends from the beginning of the declaration to the end of the declarative region; this part of the scope of a declaration is called the *immediate scope*. Furthermore, for any of the declarations in the following list, the scope of the declaration extends beyond the immediate scope:

a. A declaration that occurs immediately within a package declaration

b. An element declaration in a record type declaration

c. A formal parameter declaration in a subprogram declaration

d. A local generic declaration in a component declaration

e. A local port declaration in a component declaration

f. A formal generic declaration in an entity declaration

g. A formal port declaration in an entity declaration

In the absence of a separate subprogram declaration, the subprogram specification given in the subprogram body acts as the declaration, and rule(3) applies also in such a case. In each of these cases, the given declaration occurs immediately within some enclosing declaration, and the scope of the given declaration extends to the end of the scope of the enclosing declaration.

In addition to the above rules, the scope of any declaration that includes the end of the declarative part of a given block (whether it be an external block defined by a design entity or an internal block defined by a block statement) extends into a configuration declaration that configures the given block.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if the scope of a given declaration includes the end of the declarative part of that block, then the scope of the given declaration extends from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block. Furthermore, the scope of a use clause is similarly extended. Finally, the scope of a library unit contained within a design library is extended along with the scope of the logical library name corresponding to that design library.

NOTE--These scope rules apply to all forms of declaration. In particular, they apply also to implicit declarations.

10.3 Visibility

The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the

case of overloaded declarations, by the overloading rules. The identifiers considered in this section include any identifier other than a reserved word or attribute designator that denotes a predefined attribute. The places considered in this section are those where a lexical element (such as an identifier) occurs. The overloaded declarations considered in this section are those for subprograms and enumeration literals.

For each identifier and at each place in the text, the visibility rules determine a set of declarations (with this identifier) that define the possible meanings of an occurrence of the identifier. A declaration is said to be visible at a given place in the text when, according to the visibility rules, the declaration defines a possible meaning of this occurrence. Two cases may arise in determining the meaning of such a declaration:

-- The visibility rules determine *at most one* possible meaning. In such a case, the visibility rules are sufficient to determine the declaration defining the meaning of the occurrence of the identifier, or in the absence of such a declaration, to determine that the occurrence is not legal at the given point.

-- The visibility rules determine *more than one* possible meaning. In such a case, the occurrence of the identifier is legal at this point if and only if *exactly one* visible declaration is acceptable for the overloading rules in the given context.

A declaration is only visible within a certain part of its scope; this part starts at the end of the declaration except in the declaration of a design unit, in which case it starts immediately after the reserved word **is** is given after the identifier of the design unit. This rule applies to both explicit and implicit declarations.

Visibility is either by selection or direct. A declaration is visible *by selection* at places that are defined as follows:

a. For a primary unit contained in a library: at the place of the suffix in a selected name whose prefix denotes the library.

b. For an architecture body associated with a given entity declaration: at the place of the block specification in a block configuration for an external block whose interface is defined by that entity declaration.

c. For an architecture body associated with a given entity declaration: at the place of an architecture identifier (between the parentheses) in the first form of an entity aspect in a binding indication.

d. For a declaration given in a package declaration: at the place of the suffix in a selected name whose prefix denotes the package.

e. For an element declaration of a given record type declaration: at the place of the suffix in a selected name whose prefix is appropriate for the type; also at the place of a choice (before the compound delimiter =>) in a named element association of an aggregate of the type.

f. For a user-defined attribute: at the place of the attribute designator(after the delimiter ') in an attribute name whose prefix denotes a named entity with which that attribute has been associated.

g. For a formal parameter declaration of a given subprogram declaration: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named parameter association element of a corresponding subprogram call.

h. For a local generic declaration of a given component declaration: at the place of the formal designator in a formal part (before the compound delimiter=>) of a named generic association element of a corresponding component instantiation statement; similarly, at the place of the actual designator in an actual part (after the compound delimiter =>, if any) of a generic association element of a corresponding binding indication.

i. For a local port declaration of a given component declaration: at the place of the formal designator in a formal part

(before the compound delimiter=>) of a named port association element of a corresponding component instantiation statement; similarly, at the place of the actual designator in an actual part (after the compound delimiter =>, if any) of a port association element of a corresponding binding indication.

j. For a formal generic declaration of a given entity declaration: at the place of the formal designator in a formal part (before the compound delimiter=>) of a named generic association element of a corresponding binding indication; similarly, at the place of the formal designator in a formal part(before the compound delimiter =>) of a generic association element of a corresponding component instantiation statement when the instantiated unit is a design entity or a configuration declaration.

k. For a formal port declaration of a given entity declaration: at the place of the formal designator in a formal part (before the compound delimiter=>) of a named port association element of a corresponding binding specification; similarly, at the place of the formal designator in a formal part (before the compound delimiter =>) of a port association element of a corresponding component instantiation statement when the instantiated unit is a design entity or a configuration declaration.

l. For a formal generic declaration or a formal port declaration of a given block statement: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named association element of a corresponding generic or port map aspect.

Finally, within the declarative region associated with a construct other than a record type declaration, any declaration that occurs immediately within the region and that also occurs textually within the construct is visible by selection at the place of the suffix of an expanded name whose prefix denotes the construct.

Where it is not visible by selection, a visible declaration is said to be *directly visible*. A declaration is said to be directly visible within a certain part of its immediate scope; this part extends to the end of the immediate scope of the declaration but excludes places where the declaration is hidden as explained in the following paragraphs. In addition, a declaration occurring immediately within the visible part of a package can be made directly visible by means of a use clause according to the rules described in 10.4 .

A declaration is said to be *hidden* within (part of) an inner declarative region if the inner region contains a homograph of this declaration; the outer declaration is then hidden within the immediate scope of the inner homograph. Each of two declarations is said to be a *homograph* of the other if both declarations have the same identifier, operator symbol, or character literal, and if overloading is allowed for at most one of the two. If overloading is allowed for both declarations, then each of the two is a homograph of the other if they have the same identifier, operator symbol, or character literal, as well as the same parameter and result type profile (see 3.1.1).

Within the specification of a subprogram, every declaration with the same designator as the subprogram is hidden. Where hidden in this manner, a declaration is visible neither by selection nor directly.

Two declarations that occur immediately within the same declarative region must not be homographs, unless exactly one of them is the implicit declaration of a predefined operation. In such cases, a predefined operation is always hidden by the other homograph. Where hidden in this manner, an implicit declaration is hidden within the entire scope of the other declaration (regardless of which declaration occurs first); the implicit declaration is visible neither by selection nor directly.

Whenever a declaration with a certain identifier is visible from a given point, the identifier and the named entity (if any) are also said to be visible from that point. Direct visibility and visibility by selection are likewise defined for character literals and operator symbols. An operator is directly visible if and only if the corresponding operator declaration is directly visible.

In addition to the above rules, any declaration that is visible by selection at the end of the declarative part of a given (external or internal) block is visible by selection in a configuration declaration that configures the given block.

In addition, any declaration that is directly visible at the end of the declarative part of a given block is directly visible in a

block configuration that configures the given block. This rule holds unless a use clause that makes a homograph of the declaration potentially visible (see 10.4) appears in the corresponding configuration declaration, and if the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the declaration will be directly visible within the corresponding configuration items, except at those places that fall within the scope of the additional use clause. At such places, neither name will be directly visible.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if a given declaration is visible by selection at the end of the declarative part of that block, then the given declaration is visible by selection from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if a given declaration is directly visible at the end of the declarative part of that block, then the given declaration is visible by selection from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block. Furthermore, the visibility of declarations made directly visible by a use clause within a block is similarly extended. Finally, the visibility of a logical library name corresponding to a design library directly visible at the end of a block is similarly extended. The rules of this paragraph hold unless a use clause that makes a homograph of the declaration potentially visible appears in the corresponding block configuration, and if the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the declaration will be directly visible within the corresponding configuration items, except at those places that fall within the scope of the additional use clause. At such places, neither name will be directly visible.

NOTES

1--The same identifier, character literal, or operator symbol may occur indifferent declarations and may thus be associated with different named entities, even if the scopes of these declarations overlap. Overlap of the scopes of declarations with the same identifier, character literal, or operator symbol can result from overloading of subprograms and of enumeration literals. Such overlaps can also occur for named entities declared in the visible parts of packages and for formal generics and ports, record elements, and formal parameters, where there is overlap of the scopes of the enclosing package declarations, entity interfaces, record type declarations, or subprogram declarations. Finally, overlapping scopes can result from nesting.

2--The rules defining immediate scope, hiding, and visibility imply that a reference to an identifier, character literal, or operator symbol within its own declaration is illegal (except for design units). The identifier,character literal, or operator symbol hides outer homographs within its immediate scope--that is, from the start of the declaration. On the other hand, the identifier, character literal, or operator symbol is visible only after the end of the declaration (again, except for design units). For this reason, all but the last of the following declarations are illegal:

```
          constant K: INTEGER := K*K;              --  Illegal
          constant T: T;                           --  Illegal
          procedure P (X: P);                      --  Illegal
          function Q (X: REAL := Q) return Q;      --  Illegal
          procedure R (R: REAL);                   --  Legal (although perhaps
confusing)
```

Example:

```
   L1:  block
              signal A,B: Bit ;
        begin
        L2:  block
                 signal B: Bit ;                          --  An inner homograph
```

```
of B.
            begin
                A <= B after 5 ns;                          --  Means L1.A <= L2.B
                B <= L1.B after 10 ns;                      --  Means L2.B <= L1.B
            end block ;
                B <= A after 15 ns;                         --  Means L1.B <= L1.A
        end block ;
```

10.4 Use clauses

A use clause achieves direct visibility of declarations that are visible by selection.

```
use_clause ::=
    use selected_name { , selected_name } ;
```

Each selected name in a use clause identifies one or more declarations that will potentially become directly visible. If the suffix of the selected name is a simple name, character literal, or operator symbol, then the selected name identifies only the declaration(s) of that simple name, character literal, or operator symbol contained within the package or library denoted by the prefix of the selected name. If the suffix is the reserved word **all**, then the selected name identifies all declarations that are contained within the package or library denoted by the prefix of the selected name.

For each use clause, there is a certain region of text called the *scope* of the use clause. This region starts immediately after the use clause. If a use clause is a declarative item of some declarative region, the scope of the clause extends to the end of the declarative region. If a use clause occurs within the context clause of a design unit, the scope of the use clause extends to the end of the declarative region associated with the design unit. The scope of a use clause may additionally extend into a configuration declaration(see 10.2).

In order to determine which declarations are made directly visible at a given place by use clauses, consider the set of declarations identified by all use clauses whose scopes enclose this place. Any declaration in this set is a potentially visible declaration. A potentially visible declaration is actually made directly visible except in the following two cases:

a. A potentially visible declaration is not made directly visible if the place considered is within the immediate scope of a homograph of the declaration.

b. Potentially visible declarations that have the same designator are not made directly visible unless each of them is either an enumeration literal specification or the declaration of a subprogram (either by a subprogram declaration or by an implicit declaration).

NOTES

1--These rules guarantee that a declaration that is made directly visible by a use clause cannot hide an otherwise directly visible declaration.

2--If a named entity X declared in package P is made potentially visible within a package Q (e.g., by the inclusion of the clause "**use** P.X;" in the context clause of package Q), and the context clause for design unit R includes the clause "**use** Q.**all**;", this does not imply that X will be potentially visible in R. Only those named entities that are actually declared in package Q will be potentially visible in design unit R (in the absence of any other use clauses).

10.5 The context of overload resolution

Overloading is defined for names, subprograms, and enumeration literals.

For overloaded entities, overload resolution determines the actual meaning that an occurrence of an identifier or a character literal has whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence; overload resolution likewise determines the actual meaning of an occurrence of an operator or basic operation (see the introduction to Section 3).

At such a place, all visible declarations are considered. The occurrence is only legal if there is exactly one interpretation of each constituent of the innermost complete context; a *complete context* is either a declaration,a specification, or a statement.

When considering possible interpretations of a complete context, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below.

a. Any rule that requires a name or expression to have a certain type or to have the same type as another name or expression.

b. Any rule that requires the type of a name or expression to be a type of a certain class; similarly, any rule that requires a certain type to be a discrete, integer, floating point, physical, universal, character, or Boolean type.

c. Any rule that requires a prefix to be appropriate for a certain type.

d. The rules that require the type of an aggregate or string literal to be determinable solely from the enclosing complete context. Similarly, the rules that require the type of the prefix of an attribute, the type of the expression of a case statement, or the type of the operand of a type conversion to be determinable independently of the context.

e. The rules given for the resolution of overloaded subprogram calls; for the implicit conversions of universal expressions; for the interpretation of discrete ranges with bounds having a universal type; and for the interpretation of an expanded name whose prefix denotes a subprogram.

f. The rules given for the requirements on the return type, the number of formal parameters, and the types of the formal parameters of the subprogram denoted by the resolution function name (see 2.4).

NOTES

1--If there is only one possible interpretation of an occurrence of an identifier, character literal, operator symbol, or string, that occurrence denotes the corresponding named entity. However, this condition does not mean that the occurrence is necessarily legal since other requirements exist that are not considered for overload resolution: for example, the fact that the expression is static, the parameter modes, conformance rules, the use of named association in an indexed name, the use of **open** in an indexed name, the use of a slice as an actual to a function call, and so forth.

2--A loop parameter specification is a declaration, and hence a complete context.

3--Rules that require certain constructs to have the same parameter and result type profile fall under category a. The same holds for rules that require conformance of two constructs, since conformance requires that corresponding names be given the same meaning by the visibility and overloading rules.

Section 8

Sequential statements

The various forms of sequential statements are described in this section. Sequential statements are used to define algorithms for the execution of a subprogram or process; they execute in the order in which they appear.

```
sequence_of_statements ::=
    { sequential_statement }

sequential_statement ::=
     wait_statement
   | assertion_statement
   | report_statement
   | signal_assignment_statement
   | variable_assignment_statement
   | procedure_call_statement
   | if_statement
   | case_statement
   | loop_statement
   | next_statement
   | exit_statement
   | return_statement
   | null_statement
```

All sequential statements may be labeled. Such labels are implicitly declared at the beginning of the declarative part of the innermost enclosing process statement or subprogram body.

8.1 Wait statement

The wait statement causes the suspension of a process statement or a procedure.

```
wait_statement ::=
    [ label : ] wait [ sensitivity_clause ] [ condition_clause ] [
timeout_clause ] ;

sensitivity_clause ::=  on sensitivity_list

sensitivity_list ::=  signal_name { , signal_name }

condition_clause ::=  until condition

condition ::=  boolean_expression

timeout_clause ::=  for time_expression
```

The sensitivity clause defines the *sensitivity set* of the wait statement,which is the set of signals to which the wait statement is sensitive. Each signal name in the sensitivity list identifies a given signal as a member of the sensitivity set. Each signal name in the sensitivity list must be a static signal name, and each name must denote a signal for which reading is permitted. If no

sensitivity clause appears, the sensitivity set is constructed according to the following (recursive) rule:

The sensitivity set is initially empty. For each primary in the condition of the condition clause, if the primary is

-- A simple name that denotes a signal, add the longest static prefix of the name to the sensitivity set

-- A selected name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set

-- An expanded name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set

-- An indexed name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set and apply this rule to all expressions in the indexed name

-- A slice name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set and apply this rule to any expressions appearing in the discrete range of the slice name

-- An attribute name, if the designator denotes a signal attribute, add the longest static prefix of the name of the implicit signal denoted by the attribute name to the sensitivity set; otherwise, apply this rule to the prefix of the attribute name

-- An aggregate, apply this rule to every expression appearing after the choices and the =>, if any, in every element association

-- A function call, apply this rule to every actual designator in every parameter association

-- An actual designator of **open** in a parameter association, do not add to the sensitivity set

-- A qualified expression, apply this rule to the expression or aggregate qualified by the type mark, as appropriate

-- A type conversion, apply this rule to the expression type converted by the type mark

-- A parenthesized expression, apply this rule to the expression enclosed within the parentheses

-- Otherwise, do not add to the sensitivity set

This rule is also used to construct the sensitivity sets of the wait statements in the equivalent process statements for concurrent procedure call statements(9.3), concurrent assertion statements (9.4), and concurrent signal assignment statements (9.5).

If a signal name that denotes a signal of a composite type appears in a sensitivity list, the effect is as if the name of each scalar subelement of that signal appears in the list.

The condition clause specifies a condition that must be met for the process to continue execution. If no condition clause appears, the condition clause **until** TRUE is assumed.

The timeout clause specifies the maximum amount of time the process will remain suspended at this wait statement. If no timeout clause appears, the timeout clause **for** (STD.STANDARD.TIME'HIGH - STD.STANDARD.NOW) is assumed. It is an error if the time expression in the timeout clause evaluates to a negative value.

The execution of a wait statement causes the time expression to be evaluated to determine the *timeout interval*. It also causes the execution of the corresponding process statement to be suspended, where the corresponding process statement is the one that either contains the wait statement or is the parent (see 2.2) of the procedure that contains the wait statement. The suspended process will resume, at the latest, immediately after the timeout interval has expired.

The suspended process may also resume as a result of an event occurring on any signal in the sensitivity set of the wait

statement. If such an event occurs,the condition in the condition clause is evaluated. If the value of the condition is TRUE, the process will resume. If the value of the condition is FALSE, the process will re-suspend. Such re-suspension does not involve the recalculation of the timeout interval.

It is an error if a wait statement appears in a function subprogram or in a procedure that has a parent that is a function subprogram. Furthermore, it is an error if a wait statement appears in an explicit process statement that includes a sensitivity list or in a procedure that has a parent that is such a process statement.

Example:

```
type Arr is array (1 to 5) of BOOLEAN;
function F (P: BOOLEAN) return BOOLEAN;
signal S: Arr;
signal l, r: INTEGER range 1 to 5;

--  The following two wait statements have the same meaning:

wait until F(S(3)) and (S(l) or S(r));
wait on S(3), S, l, r until F(S(3)) and (S(l) or S(r));
```

NOTES

1--The wait statement **wait until** Clk = '1'; has semantics identical to

```
loop
     wait on Clk;
     exit when Clk = '1';
end loop;
```

because of the rules for the construction of the default sensitivity clause. These same rules imply that **wait until** True; has semantics identical to **wait**;.

2--The conditions that cause a wait statement to resume execution of its enclosing process may no longer hold at the time the process resumes execution if the enclosing process is a postponed process.

3--The rule for the construction of the default sensitivity set implies that if a function call appears in a condition clause and the called function is an impure function,then any signals that are accessed by the function but that are not passed through the association list of the call are not added to the default sensitivity set for the condition by virtue of the appearance of the function call in the condition.

8.2 Assertion statement

An assertion statement checks that a specified condition is true and reports an error if it is not.

```
assertion_statement ::=  [ label : ] assertion ;

assertion ::=
     assert condition
         [ report expression ]
         [ severity expression ]
```

If the **report** clause is present, it must include an expression of predefined type STRING that specifies a message to be reported. If the **severity** clause is present, it must specify an expression of predefined type SEVERITY_LEVEL that specifies the severity level of the assertion.

The **report** clause specifies a message string to be included in error messages generated by the assertion. In the absence of a **report** clause for a given assertion, the string "Assertion violation." is the default value for the message string. The **severity** clause specifies a severity level associated with the assertion. In the absence of a **severity** clause fora given assertion, the default value of the severity level is ERROR.

Evaluation of an assertion statement consists of evaluation of the Boolean expression specifying the condition. If the expression results in the value FALSE, then an *assertion violation* is said to occur. When an assertion violation occurs, the **report** and **severity** clause expressions of the corresponding assertion, if present, are evaluated. The specified message string and severity level (or the corresponding default values, if not specified) are then used to construct an error message.

The error message consists of at least

a. An indication that this message is from an assertion
b. The value of the severity level
c. The value of the message string
d. The name of the design unit (see 11.1) containing the assertion

8.3 Report statement

A report statement displays a message.

```
report_statement ::=
    [ label : ]
        report expression
            [ severity expression ] ;
```

The **report** statement expression must be of the predefined type STRING. The string value of this expression is included in the message generated by the report statement. If the **severity** clause is present, it must specify an expression of predefined type SEVERITY_LEVEL. The severity clause specifies a severity level associated with the report. In the absence of a **severity** clause for a given report, the default value of the severity level is NOTE.

The evaluation of a report statement consists of the evaluation of the report expression and severity clause expression, if present. The specified message string and severity level (or corresponding default, if the severity level is not specified) are then used to construct a report message.

The report message consists of at least

a. An indication that this message is from a report statement
b. The value of the severity level
c. The value of the message string
d. The name of the design unit containing the report statement

Example:

```
    report "Entering process P";                              --  A report statement
                                                              --  with default
severity NOTE.

    report "Setup or Hold violation; outputs driven to 'X'    --  Another report
statement;
         severity WARNING;                                        --  severity is
specified.
```

8.4 Signal assignment statement

A signal assignment statement modifies the projected output waveforms contained in the drivers of one or more signals (see 12.6.1).

```
signal_assignment_statement ::=
    [ label : ] target <= [ delay_mechanism ] waveform ;

delay_mechanism ::=
      transport
  | [ reject time_expression ] inertial

target ::=
      name
   | aggregate

waveform ::=
     waveform_element { , waveform_element }
   | unaffected
```

If the target of the signal assignment statement is a name, then the name must denote a signal, and the base type of the value component of each transaction produced by a waveform element on the right-hand side must be the same as the base type of the signal denoted by that name. This form of signal assignment assigns right-hand side values to the drivers associated with a single (scalar or composite) signal.

If the target of the signal assignment statement is in the form of an aggregate, then the type of the aggregate must be determinable from the context, excluding the aggregate itself but including the fact that the type of the aggregate must be a composite type. The base type of the value component of each transaction produced by a waveform element on the right-hand side must be the same as the base type of the aggregate. Furthermore, the expression in each element association of the aggregate must be a locally static name that denotes a signal. This form of signal assignment assigns slices or subelements of the right-hand side values to the drivers associated with the signal named as the corresponding slice or subelement of the aggregate.

If the target of a signal assignment statement is in the form of an aggregate,and if the expression in an element association of that aggregate is a signal name that denotes a given signal, then the given signal and each subelement thereof (if any) are said to be *identified* by that element association as targets of the assignment statement. It is an error if a given signal or any subelement thereof is identified as a target by more than one element association in such an aggregate. Furthermore, it is an error if an element association in such an aggregate contains an **others** choice or a choice that is a discrete range.

The right-hand side of a signal assignment may optionally specify a delay mechanism. A delay mechanism consisting of the reserved word **transport** specifies that the delay associated with the first waveform element is to be construed as *transport* delay. Transport delay is characteristic of hardware devices (such as transmission lines) that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. If no delay mechanism is present, or if a delay mechanism including the reserved word **inertial** is present, the delay is construed to be *inertial* delay. Inertial delay is characteristic of switching circuits:a pulse whose duration is shorter than the switching time of the circuit will not be transmitted, or in the case that a pulse rejection limit is specified, a pulse whose duration is shorter than that limit will not be transmitted.

Every inertially delayed signal assignment has a *pulse rejection limit*. If the delay mechanism specifies inertial delay, and if the reserved word **reject** followed by a time expression is present, then the time expression specifies the pulse rejection limit. In all other cases, the pulse rejection limit is specified by the time expression associated with the first waveform element.

It is an error if the pulse rejection limit for any inertially delayed signal assignment statement is either negative or greater than the time expression associated with the first waveform element.

It is an error if the reserved word **unaffected** appears as a waveform in a (sequential) signal assignment statement.

NOTE--The reserved word **unaffected** may only appear as a waveform in concurrent signal assignment statements. See 9.5.1 .

Examples:

```
--  Assignments using inertial delay:

       --  The following three assignments are equivalent to each other:

            Output_pin <= Input_pin after 10 ns;
            Output_pin <= inertial Input_pin after 10 ns;
            Output_pin <= reject 10 ns inertial Input_pin after 10 ns;

       --  Assignments with a pulse rejection limit less than the time expression:

            Output_pin <= reject 5 ns inertial Input_pin after 10 ns;
            Output_pin <= reject 5 ns inertial Input_pin after 10 ns, not Input_pin
after 20 ns;

--  Assignments using transport delay:

            Output_pin <= transport Input_pin after 10 ns;
            Output_pin <= transport Input_pin after 10 ns, not Input_pin after 20 ns;

       --  Their equivalent assignments:

            Output_pin <= reject 0 ns inertial Input_pin after 10 ns;
            Output_pin <= reject 0 ns inertial Input_pin after 10 ns, not Input_pin
after 10 ns;
```

NOTE--If a right-hand side value expression is either a numeric literal or an attribute that yields a result of type *universal_integer* or *universal_real*,then an implicit type conversion is performed.

8.4.1 Updating a projected output waveform

The effect of execution of a signal assignment statement is defined in terms of its effect upon the projected output waveforms (see 12.6.1) representing the current and future values of drivers of signals.

```
waveform_element ::=
     value_expression [ after time_expression ]
   | null [ after time_expression ]
```

The future behavior of the driver(s) for a given target is defined by transactions produced by the evaluation of waveform elements in the waveform of a signal assignment statement. The first form of waveform element is used to specify that the driver is to assign a particular value to the target at the specified time. The second form of waveform element is used to specify that the driver of the signal is to be turned off, so that it (at least temporarily) stops contributing to the value of the target. This form of waveform element is called a *null waveform element*. It is an error if the target of a signal assignment statement containing a null waveform element is not a guarded signal or an aggregate of guarded signals.

The base type of the time expression in each waveform element must be the predefined physical type TIME as defined in package STANDARD. If the **after** clause of a waveform element is not present, then an implicit"**after** 0 ns" is assumed. It is an error if the time expression in a waveform element evaluates to a negative value.

Evaluation of a waveform element produces a single transaction. The time component of the transaction is determined by the

current time added to the value of the time expression in the waveform element. For the first form of waveform element, the value component of the transaction is determined by the value expression in the waveform element. For the second form of waveform element, the value component is not defined by the language, but it is defined to be of the type of the target. A transaction produced by the evaluation of the second form of waveform element is called a *null transaction*.

For the execution of a signal assignment statement whose target is of a scalar type, the waveform on its right-hand side is first evaluated. Evaluation of a waveform consists of the evaluation of each waveform element in the waveform. Thus, the evaluation of a waveform results in a sequence of transactions, where each transaction corresponds to one waveform element in the waveform. These transactions are called *new* transactions. It is an error if the sequence of new transactions is not in ascending order with respect to time.

The sequence of transactions is then used to update the projected output waveform representing the current and future values of the driver associated with the signal assignment statement. Updating a projected output waveform consists of the deletion of zero or more previously computed transactions(called *old* transactions) from the projected output waveform and the addition of the new transactions, as follows:

a. All old transactions that are projected to occur at or after the time at which the earliest new transaction is projected to occur are deleted from the projected output waveform.

b. The new transactions are then appended to the projected output waveform in the order of their projected occurrence.

   If the initial delay is inertial delay according to the definitions of 8.4 , the projected output waveform is further modified as follows:

c. All of the new transactions are marked.

d. An old transaction is marked if the time at which it is projected to occur is less than the time at which the first new transaction is projected to occur minus the pulse rejection limit.

e. For each remaining unmarked, old transaction, the old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction.

f. The transaction that determines the current value of the driver is marked.

g. All unmarked transactions (all of which are old transactions) are deleted from the projected output waveform.

For the purposes of marking transactions, any two successive null transactions in a projected output waveform are considered to have the same value component.

The execution of a signal assignment statement whose target is of a composite type proceeds in a similar fashion, except that the evaluation of the waveform results in one sequence of transactions for each scalar subelement of the type of the target. Each such sequence consists of transactions whose value portions are determined by the values of the same scalar subelement of the value expressions in the waveform, and whose time portion is determined by the time expression corresponding to that value expression. Each such sequence is then used to update the projected output waveform of the driver of the matching subelement of the target. This applies both to a target that is the name of a signal of a composite type and to a target that is in the form of an aggregate.

If a given procedure is declared by a declarative item that is not contained within a process statement, and if a signal assignment statement appears in that procedure, then the target of the assignment statement must be a formal parameter of the given procedure or of a parent of that procedure, or an aggregate of such formal parameters. Similarly, if a given procedure is declared by a declarative item that is not contained within a process statement, and if a signal is associated with an **inout** or **out** mode signal parameter in a subprogram call within that procedure, then the signal so associated must be a formal parameter of the given procedure or of a parent of that procedure.

NOTES

1--These rules guarantee that the driver affected by a signal assignment statement is always statically determinable if the signal assignment appears within a given process (including the case in which it appears within a procedure that is declared within the given process). In this case, the affected driver is the one defined by the process; otherwise, the signal assignment must appear within a procedure, and the affected driver is the one passed to the procedure along with a signal parameter of that procedure.

2--Overloading the operator "=" has no effect on the updating of a projected output waveform.

3--Consider a signal assignment statement of the form

```
T <= reject tr inertial e1 after t1 { , ei after ti<./sub> } ;
```

The following relations hold:

$0\ ns <= t_r <= t_1$

and

$0\ ns <= t_i < t_i+1$

Note that, if $t_r = 0$ ns, then the waveform editing is identical to that for transport-delayed assignment, and if $t_r = t_1$, the waveform is identical to that for the statement

```
T <= e1 after t1 { , ei after ti } ;
```

4--Consider the following signal assignment in some process:

```
S <= reject 15 ns inertial 12 after 20 ns, 18 after 41 ns;
```

where S is a signal of some integer type. Assume that at the time this signal assignment is executed, the driver of S in the process has the following contents (the first entry is the current driving value):

1	2	2	12	5	8
NOW	+3 ns	+12 ns	+13 ns	+20 ns	+42 ns

(The times given are relative to the current time.) The updating of the projected output waveform proceeds as follows:

a. The driver is truncated at 20 ns. The driver now contains the following pending transactions:

1	2	2	12
NOW	+3 ns	+12 ns	+13 ns

b. The new waveforms are added to the driver. The driver now contains the following pending transactions:

1	2	2	12	12	18
NOW	+3 ns	+12 ns	+13 ns	+20 ns	+41 ns

c. All new transactions are marked, as well as those old transactions that occur at less than the time of the first new waveform (20 ns) less the rejection limit (15 ns). The driver now contains the following pending transactions (marked transactions are emboldened):

1	**2**	2	12	**12**	**18**
NOW	**+3 ns**	+12 ns	+13 ns	**+20 ns**	**+41 ns**

d. Each remaining unmarked transaction is marked if it immediately precedes a marked transaction and has the same value as the marked transaction. The driver now contains the following pending transactions:

1	**2**	2	**12**	**12**	**18**
NOW	**+3 ns**	+12 ns	**+13 ns**	**+20 ns**	**+41 ns**

e. The transaction that determines the current value of the driver is marked, and all unmarked transactions are then deleted. The final driver contents are then as follows, after clearing the markings:

1	2	12	12	18
NOW	+3 ns	+13 ns	+20 ns	+41 ns

5--No subtype check is performed on the value component of a new transaction when it is added to a driver. Instead, a subtype check that the value component of a transaction belongs to the subtype of the signal driven by the driver is made when the driver takes on that value. See 12.6.1 .

8.5 Variable assignment statement

A variable assignment statement replaces the current value of a variable with anew value specified by an expression. The named variable and the right-hand side expression must be of the same type.

```
variable_assignment_statement ::=
    [ label : ] target := expression ;
```

If the target of the variable assignment statement is a name, then the name must denote a variable, and the base type of the expression on the right-hand side must be the same as the base type of the variable denoted by that name. This form of variable assignment assigns the right-hand side value to a single(scalar or composite) variable.

If the target of the variable assignment statement is in the form of an aggregate, then the type of the aggregate must be determinable from the context, excluding the aggregate itself but including the fact that the type of the aggregate must be a composite type. The base type of the expression on the right-hand side must be the same as the base type of the aggregate. Furthermore, the expression in each element association of the aggregate must be a locally static name that denotes a variable. This form of variable assignment assigns each subelement or slice of the right-hand side value to the variable named as the corresponding subelement or slice of the aggregate.

If the target of a variable assignment statement is in the form of an aggregate, and if the locally static name in an element association of that aggregate denotes a given variable or denotes another variable of which the given variable is a subelement or slice, then the element association is said to *identify* the given variable as a target of the assignment statement. It is an error if a given variable is identified as a target by more than one element association in such an aggregate.

For the execution of a variable assignment whose target is a variable name, the variable name and the expression are first evaluated. A check is then made that the value of the expression belongs to the subtype of the variable, except in the case of a variable that is an array (in which case the assignment involves a subtype conversion). Finally, the value of the expression becomes the new value of the variable. A design is erroneous if it depends on the order of evaluation of the target and source expressions of an assignment statement.

The execution of a variable assignment whose target is in the form of an aggregate proceeds in a similar fashion, except that each of the names in the aggregate is evaluated, and a subtype check is performed for each subelement or slice of the right-hand side value that corresponds to one of the names in the aggregate. The value of the subelement or slice of the right-hand side value then becomes the new value of the variable denoted by the corresponding name.

An error occurs if the aforementioned subtype checks fail.

The determination of the type of the target of a variable assignment statement may require determination of the type of the expression if the target is a name that can be interpreted as the name of a variable designated by the access value returned by a function call, and similarly, as an element or slice of such a variable.

NOTE--If the right-hand side is either a numeric literal or an attribute that yields a result of type universal integer or universal real, then an implicit type conversion is performed.

8.5.1 Array variable assignments

If the target of an assignment statement is a name denoting an array variable(including a slice), the value assigned to the target is implicitly converted to the subtype of the array variable; the result of this subtype conversion becomes the new value of the array variable.

This means that the new value of each element of the array variable is specified by the matching element (see 7.2.2) in the corresponding array value obtained by evaluation of the expression. The subtype conversion checks that for each element of the array variable there is a matching element in the array value, and vice versa. An error occurs if this check fails.

NOTE--The implicit subtype conversion described for assignment to an array variable is performed only for the value of the right-hand side expression as a whole; it is not performed for subelements or slices that are array values.

8.6 Procedure call statement

A procedure call invokes the execution of a procedure body.

```
procedure_call_statement ::=  [ label : ] procedure_call ;

procedure_call ::=  procedure_name [ ( actual_parameter_part ) ]
```

The procedure name specifies the procedure body to be invoked. The actual parameter part, if present, specifies the association of actual parameters with formal parameters of the procedure.

For each formal parameter of a procedure, a procedure call must specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element (other than the actual **open**) in the association list or, in the absence of such an association element, by a default expression (see 4.3.2).

Execution of a procedure call includes evaluation of the actual parameter expressions specified in the call and evaluation of the default expressions associated with formal parameters of the procedure that do not have actual parameters associated with them. In both cases, the resulting value must belong to the subtype of the associated formal parameter. (If the formal parameter is of an unconstrained array type, then the formal parameter takes on the subtype of the actual parameter.) The procedure body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

8.7 If statement

An if statement selects for execution one or none of the enclosed sequences of statements, depending on the value of one or more corresponding conditions.

```
if_statement ::=
     [ if_label : ]
           if condition then
                sequence_of_statements
           { elsif condition then
                sequence_of_statements }
           [ else
                sequence_of_statements ]
           end if [ if_label ] ;
```

If a label appears at the end of an if statement, it must repeat the if label.

For the execution of an if statement, the condition specified after **if**,and any conditions specified after **elsif**, are evaluated in succession(treating a final **else** as **elsif** TRUE **then**) until one evaluates to TRUE or all conditions are evaluated and yield FALSE. If one condition evaluates to TRUE, then the corresponding sequence of statements is executed; otherwise, none of the sequences of statements is executed.

8.8 Case statement

A case statement selects for execution one of a number of alternative sequences of statements; the chosen alternative is defined by the value of an expression.

```
case_statement ::=
    [ case_label : ]
         case expression is
             case_statement_alternative
             { case_statement_alternative }
         end case [ case_label ] ;

case_statement_alternative ::=
    when choices =>
       sequence_of_statements
```

The expression must be of a discrete type, or of a one-dimensional array type whose element base type is a character type. This type must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type or a one-dimensional character array type. Each choice in a case statement alternative must be of the same type as the expression; the list of choices specifies for which values of the expression the alternative is chosen.

If the expression is the name of an object whose subtype is locally static,whether a scalar type or an array type, then each value of the subtype must be represented once and only once in the set of choices of the case statement, and no other value is allowed; this rule is likewise applied if the expression is a qualified expression or type conversion whose type mark denotes a locally static subtype, or if the expression is a call to a function whose return type mark denotes a locally static subtype.

If the expression is of a one-dimensional character array type, then the expression must be one of the following:

-- The name of an object whose subtype is locally static

-- An indexed name whose prefix is one of the members of this list and whose indexing expressions are locally static expressions

-- A slice name whose prefix is one of the members of this list and whose discrete range is a locally static discrete range

-- A function call whose return type mark denotes a locally static subtype

-- A qualified expression or type conversion whose type mark denotes a locally static subtype

In such a case, each choice appearing in any of the case statement alternatives must be a locally static expression whose value is of the same length as that of the case expression. It is an error if the element subtype of the one-dimensional character array type is not a locally static subtype.

For other forms of expression, each value of the (base) type of the expression must be represented once and only once in the set of choices, and no other value is allowed.

The simple expression and discrete ranges given as choices in a case statement must be locally static. A choice defined by a discrete range stands for all values in the corresponding range. The choice **others** is only allowed for the last alternative and as its only choice; it stands for all values(possibly none) not given in the choices of previous alternatives. An element simple name (see 7.3.2) is not allowed as a choice of a case statement alternative.

If a label appears at the end of a case statement, it must repeat the case label.

The execution of a case statement consists of the evaluation of the expression followed by the execution of the chosen sequence of statements.

NOTES

1--The execution of a case statement chooses one and only one alternative,since the choices are exhaustive and mutually exclusive. A qualified expression whose type mark denotes a locally static subtype can often be used as the expression of a case statement to limit the number of choices that need be explicitly specified.

2--An **others** choice is required in a case statement if the type of the expression is the type *universal_integer* (for example, if the expression is an integer literal), since this is the only way to cover all values of the type *universal_integer*.

3--Overloading the operator "=" has no effect on the semantics of case statement execution.

8.9 Loop statement

A loop statement includes a sequence of statements that is to be executed repeatedly, zero or more times.

```
loop_statement ::=
    [ loop_label : ]
         [ iteration_scheme ] loop
              sequence_of_statements
         end loop [ loop_label ] ;

iteration_scheme ::=
       while condition
    | for loop_parameter_specification

parameter_specification ::=
    identifier in discrete_range
```

If a label appears at the end of a loop statement, it must repeat the label at the beginning of the loop statement.

Execution of a loop statement is complete when the loop is left as a consequence of the completion of the iteration scheme (see below), if any, or the execution of a next statement, an exit statement, or a return statement.

A loop statement without an iteration scheme specifies repeated execution of the sequence of statements.

For a loop statement with a **while** iteration scheme, the condition is evaluated before each execution of the sequence of statements; if the value of the condition is TRUE, the sequence of statements is executed; if FALSE, the iteration scheme is said to be *complete* and the execution of the loop statement is complete.

For a loop statement with a **for** iteration scheme, the loop parameter specification is the declaration of the *loop parameter* with the given identifier. The loop parameter is an object whose type is the base type of the discrete range. Within the sequence of statements, the loop parameter is a constant. Hence, a loop parameter is not allowed as the target of an assignment statement. Similarly, the loop parameter must not be given as an actual corresponding to a formal of mode **out** or **inout** in an association list.

For the execution of a loop with a **for** iteration scheme, the discrete range is first evaluated. If the discrete range is a null range, the iteration scheme is said to be *complete* and the execution of the loop statement is therefore complete; otherwise, the sequence of statements is executed once for each value of the discrete range (subject to the loop not being left as a consequence of the execution of a next statement, an exit statement, or a return statement), after which the iteration scheme is said to be *complete*. Prior to each such iteration, the corresponding value of the discrete range is assigned to the loop parameter. These values are assigned in left-to-right order.

NOTE--A loop may be left as the result of the execution of a next statement if the loop is nested inside of an outer loop and the next statement has a loop label that denotes the outer loop.

8.10 Next statement

A next statement is used to complete the execution of one of the iterations of an enclosing loop statement (called "loop" in the following text). The completion is conditional if the statement includes a condition.

```
next_statement ::=
    [ label : ] next [ loop_label ] [ when condition ] ;
```

A next statement with a loop label is only allowed within the labeled loop and applies to that loop; a next statement without a loop label is only allowed within a loop and applies only to the innermost enclosing loop (whether labeled or not).

For the execution of a next statement, the condition, if present, is first evaluated. The current iteration of the loop is terminated if the value of the condition is TRUE or if there is no condition.

8.11 Exit statement

An exit statement is used to complete the execution of an enclosing loop statement (called "loop" in the following text). The completion is conditional if the statement includes a condition.

```
exit_statement ::=
    [ label : ] exit [ loop_label ] [ when condition ] ;
```

An exit statement with a loop label is only allowed within the labeled loop and applies to that loop; an exit statement without a loop label is only allowed within a loop and applies only to the innermost enclosing loop (whether labeled or not).

For the execution of an exit statement, the condition, if present, is first evaluated. Exit from the loop then takes place if the value of the condition is TRUE or if there is no condition.

8.12 Return statement

A return statement is used to complete the execution of the innermost enclosing function or procedure body.

```
return_statement ::=
    [ label : ] return [ expression ] ;
```

A return statement is only allowed within the body of a function or procedure,and it applies to the innermost enclosing function or procedure.

A return statement appearing in a procedure body must not have an expression. A return statement appearing in a function body must have an expression.

The value of the expression defines the result returned by the function. The type of this expression must be the base type of the type mark given after the reserved word **return** in the specification of the function. It is an error if execution of a function completes by any means other than the execution of a return statement.

For the execution of a return statement, the expression (if any) is first evaluated and a check is made that the value belongs to the result subtype. The execution of the return statement is thereby completed if the check succeeds; so also is the execution of the enclosing subprogram. An error occurs at the place of the return statement if the check fails.

NOTES

1--If the expression is either a numeric literal, or an attribute that yields a result of type *universal_integer* or *universal_real*, then an implicit conversion of the result is performed.

2--If the return type mark of a function denotes a constrained array subtype,then no implicit subtype conversions are performed on the values of the expressions of the return statements within the subprogram body of that function. Thus, for each index position of each value, the bounds of the discrete range must be the same as the discrete range of the return subtype,and the directions must be the same.

8.13 Null statement

A null statement performs no action.

```
null_statement ::=
    [ label : ] null ;
```

The execution of the null statement has no effect other than to pass on to the next statement.

NOTE--The null statement can be used to specify explicitly that no action is to be performed when certain conditions are true, although it is never mandatory for this (or any other) purpose. This is particularly useful in conjunction with the case statement, in which all possible values of the case expression must be covered by choices: for certain choices, it may be that no action is required.

Section 12

Elaboration and execution

The process by which a declaration achieves its effect is called the *elaboration* of the declaration. After its elaboration, a declaration is said to be elaborated. Prior to the completion of its elaboration (including before the elaboration), the declaration is not yet elaborated.

Elaboration is also defined for design hierarchies, declarative parts, statement parts (containing concurrent statements), and concurrent statements. Elaboration of such constructs is necessary in order ultimately to elaborate declarative items that are declared within those constructs.

In order to execute a model, the design hierarchy defining the model must first be elaborated. Initialization of nets (see 12.6.2) in the model then occurs. Finally, simulation of the model proceeds. Simulation consists of the repetitive execution of the *simulation cycle*, during which processes are executed and nets updated.

12.1 Elaboration of a design hierarchy

The elaboration of a design hierarchy creates a collection of processes interconnected by nets; this collection of processes and nets can then be executed to simulate the behavior of the design.

A design hierarchy may be defined by a design entity. Elaboration of a design hierarchy defined in this manner consists of the elaboration of the block statement equivalent to the external block defined by the design entity. The architecture of this design entity is assumed to contain an implicit configuration specification (see 5.2) for each component instance that is unbound in this architecture; each configuration specification has an entity aspect denoting an anonymous configuration declaration identifying the visible entity declaration (see 5.2) and supplying an implicit block configuration (see 1.3.1) that binds and configures a design entity identified according to the rules of 5.2.2 . The equivalent block statement is defined in 9.6.2 . Elaboration of a block statement is defined in 12.4.1 .

A design hierarchy may also be defined by a configuration. Elaboration of a configuration consists of the elaboration of the block statement equivalent to the external block defined by the design entity configured by the configuration. The configuration contains an implicit component configuration(see 1.3.2) for each unbound component instance contained within the external block and an implicit block configuration (see 1.3.1) for each internal block contained within the external block.

An implementation may allow, but is not required to allow, a design entity at the root of a design hierarchy to have generics and ports. If an implementation allows these *top-level* interface objects, it may restrict their allowed types and modes in an implementation-defined manner. Similarly,the means by which top-level interface objects are associated with the external environment of the hierarchy are also defined by an implementation supporting top-level interface objects.

Elaboration of a block statement involves first elaborating each not-yet-elaborated package containing declarations referenced by the block. Similarly, elaboration of a given package involves first elaborating each not-yet-elaborated package containing declarations referenced by the given package. Elaboration of a package additionally consists of the

a. Elaboration of the declarative part of the package declaration,eventually followed by
b. Elaboration of the declarative part of the corresponding package body, if the package has a corresponding package body.

Step b above, the elaboration of a package body, may be deferred until the declarative parts of other packages have been elaborated, if necessary, because of the dependencies created between packages by their interpackage references.

Elaboration of a declarative part is defined in 12.3 .

Examples:

```
     --  In the following example, because of the dependencies between the packages,
the
     --  elaboration of either package body must follow the elaboration of both
package
     --  declarations.

     package P1 is
         constant C1: INTEGER := 42;
         constant C2: INTEGER;
     end package P1;

     package P2 is
         constant C1: INTEGER := 17;
         constant C2: INTEGER;
     end package P2;

     package body P1 is
         constant C2: INTEGER := Work.P2.C1;
     end package body P1;

     package body P2 is
         constant C2: INTEGER := Work.P1.C1;
     end package body P2;

     --  If a design hierarchy is described by the following design entity:

     entity E is end;

     architecture A of E is
         component comp
            port (...);
         end component;
     begin
     C:  comp port map (...);
     B:  block
             ...
         begin
             ...
         end block B;
     end architecture A;

     --  then its architecture contains the following implicit configuration
specification at the
     --  end of its declarative part:

         for C: comp use configuration anonymous;

     --  and the following configuration declaration is assumed to exist when E(A) is
     --  elaborated:
```

```
    configuration anonymous of L.E is                    --  L is the library in
which E(A) is found.
        for A                                            --  The most recently
analyzed architecture

                                                         --  of L.E.

        end for;
    end configuration anonymous;
```

12.2 Elaboration of a block header

Elaboration of a block header consists of the elaboration of the generic clause, the generic map aspect, the port clause, and the port map aspect, in that order.

12.2.1 The generic clause

Elaboration of a generic clause consists of the elaboration of each of the equivalent single generic declarations contained in the clause, in the order given. The elaboration of a generic declaration consists of elaborating the subtype indication and then creating a generic constant of that subtype.

The value of a generic constant is not defined until a subsequent generic map aspect is evaluated or, in the absence of a generic map aspect, until the default expression associated with the generic constant is evaluated to determine the value of the constant.

12.2.2 The generic map aspect

Elaboration of a generic map aspect consists of elaborating the generic association list. The generic association list contains an implicit association element for each generic constant that is not explicitly associated with an actual or that is associated with the reserved word **open**; the actual part of such an implicit association element is the default expression appearing in the declaration of that generic constant.

Elaboration of a generic association list consists of the elaboration of each generic association element in the association list. Elaboration of a generic association element consists of the elaboration of the formal part and the evaluation of the actual part. The generic constant or subelement or slice thereof designated by the formal part is then initialized with the value resulting from the evaluation of the corresponding actual part. It is an error if the value of the actual does not belong to the subtype denoted by the subtype indication of the formal. If the subtype denoted by the subtype indication of the declaration of the formal is a constrained array subtype,then an implicit subtype conversion is performed prior to this check. It is also an error if the type of the formal is an array type and the value of each element of the actual does not belong to the element subtype of the formal.

12.2.3 The port clause

Elaboration of a port clause consists of the elaboration of each of the equivalent single port declarations contained in the clause, in the order given. The elaboration of a port declaration consists of elaborating the subtype indication and then creating a port of that subtype.

12.2.4 The port map aspect

Elaboration of a port map aspect consists of elaborating the port association list.

Elaboration of a port association list consists of the elaboration of each port association element in the association list whose actual is not the reserved word **open**. Elaboration of a port association element consists of the elaboration of the formal part;

the port or subelement or slice thereof designated by the formal part is then associated with the signal or expression designated by the actual part. This association involves a check that the restrictions on port associations (see 1.1.1.2) are met. It is an error if this check fails.

If a given port is a port of mode **in** whose declaration includes a default expression, and if no association element associates a signal or expression with that port, then the default expression is evaluated and the effective and driving value of the port is set to the value of the default expression. Similarly, if a given port of mode **in** is associated with an expression, that expression is evaluated and the effective and driving value of the port is set to the value of the expression. In the event that the value of a port is derived from an expression in either fashion, references to the predefined attributes 'DELAYED, 'STABLE, 'QUIET, 'EVENT, 'ACTIVE, 'LAST_EVENT,'LAST_ACTIVE, 'LAST_VALUE, 'DRIVING, and 'DRIVING_VALUE of the port return values indicating that the port has the given driving value with no activity at any time (see 12.6.3).

If an actual signal is associated with a port of any mode, and if the type of the formal is a scalar type, then it is an error if (after applying any conversion function or type conversion expression present in the actual part) the bounds and direction of the subtype denoted by the subtype indication of the formal are not identical to the bounds and direction of the subtype denoted by the subtype indication of the actual. If an actual expression is associated with a formal port (of mode **in**), and if the type of the formal is a scalar type, then it is an error if the value of the expression does not belong to the subtype denoted by the subtype indication of the declaration of the formal.

If an actual signal or expression is associated with a formal port, and if the formal is of a constrained array subtype, then it is an error if the actual does not contain a matching element for each element of the formal. In the case of an actual signal, this check is made after applying any conversion function or type conversion that is present in the actual part. If an actual signal or expression is associated with a formal port, and if the subtype denoted by the subtype indication of the declaration of the formal is an unconstrained array type, then the subtype of the formal is taken from the actual associated with that formal. It is also an error if the mode of the formal is **in** or **inout** and the value of each element of the actual array (after applying any conversion function or type conversion present in the actual part) does not belong to the element subtype of the formal. If the formal port is of mode **out**, **inout**, or **buffer**, it is also an error if the value of each element of the formal(after applying any conversion function or type conversion present in the formal part) does not belong to the element subtype of the actual.

If an actual signal or expression is associated with a formal port, and if the formal is of a record subtype, then it is an error if the rules of the preceding three paragraphs do not apply to each element of the record subtype. In the case of an actual signal, these checks are made after applying any conversion function or type conversion that is present in the actual part.

12.3 Elaboration of a declarative part

The elaboration of a declarative part consists of the elaboration of the declarative items, if any, in the order in which they are given in the declarative part. This rule holds for all declarative parts, with three exceptions:

a. The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute defined in package STANDARD (see 5.1 and 14.2)
b. The architecture declarative part of a design entity whose architecture is decorated with the 'FOREIGN attribute defined in package STANDARD
c. A subprogram declarative part whose subprogram is decorated with the 'FOREIGN attribute defined in package STANDARD

For these cases, the declarative items are not elaborated; instead, the design entity or subprogram is subject to implementation-dependent elaboration.

In certain cases, the elaboration of a declarative item involves the evaluation of expressions that appear within the declarative item. The value of any object denoted by a primary in such an expression must be defined at the time the primary is read (see 4.3.2). In addition, if a primary in such an expression is a function call, then the value of any object denoted by or appearing as a part of an actual designator in the function call must be defined at the time the expression is evaluated.

NOTE--It is a consequence of this rule that the name of a signal declared within a block cannot be referenced in expressions appearing in declarative items within that block, an inner block, or process statement; nor can it be passed as a parameter to a function called during the elaboration of the block. These restrictions exist because the value of a signal is not defined until after the design hierarchy is elaborated. However, a signal parameter name maybe used within expressions in declarative items within a subprogram declarative part, provided that the subprogram is only called after simulation begins,because the value of every signal will be defined by that time.

12.3.1 Elaboration of a declaration

Elaboration of a declaration has the effect of creating the declared item.

For each declaration, the language rules (in particular scope and visibility rules) are such that it is either impossible or illegal to use a given item before the elaboration of its corresponding declaration. For example, it is not possible to use the name of a type for an object declaration before the corresponding type declaration is elaborated. Similarly, it is illegal to calla subprogram before its corresponding body is elaborated.

12.3.1.1 Subprogram declarations and bodies

Elaboration of a subprogram declaration involves the elaboration of the parameter interface list of the subprogram declaration; this in turn involves the elaboration of the subtype indication of each interface element to determine the subtype of each formal parameter of the subprogram.

Elaboration of a subprogram body has no effect other than to establish that the body can, from then on, be used for the execution of calls of the subprogram.

12.3.1.2 Type declarations

Elaboration of a type declaration generally consists of the elaboration of the definition of the type and the creation of that type. For a constrained array type declaration, however, elaboration consists of the elaboration of the equivalent anonymous unconstrained array type followed by the elaboration of the named subtype of that unconstrained type.

Elaboration of an enumeration type definition has no effect other than the creation of the corresponding type.

Elaboration of an integer, floating point, or physical type definition consists of the elaboration of the corresponding range constraint. For a physical type definition, each unit declaration in the definition is also elaborated. Elaboration of a physical unit declaration has no effect other than to create the unit defined by the unit declaration.

Elaboration of an unconstrained array type definition consists of the elaboration of the element subtype indication of the array type.

Elaboration of a record type definition consists of the elaboration of the equivalent single element declarations in the given order. Elaboration of an element declaration consists of elaboration of the element subtype indication.

Elaboration of an access type definition consists of the elaboration of the corresponding subtype indication.

12.3.1.3 Subtype declarations

Elaboration of a subtype declaration consists of the elaboration of the subtype indication. The elaboration of a subtype indication creates a subtype. If the subtype does not include a constraint, then the subtype is the same as that denoted by the type mark. The elaboration of a subtype indication that includes a constraint proceeds as follows:

a. The constraint is first elaborated.

b. A check is then made that the constraint is compatible with the type or subtype denoted by the type mark (see 3.1 and 3.2.1.1).

Elaboration of a range constraint consists of the evaluation of the range. The evaluation of a range defines the bounds and direction of the range. Elaboration of an index constraint consists of the elaboration of each of the discrete ranges in the index constraint in some order that is not defined by the language.

12.3.1.4 Object declarations

Elaboration of an object declaration that declares an object other than a file object proceeds as follows:

a. The subtype indication is first elaborated. This establishes the subtype of the object.
b. If the object declaration includes an explicit initialization expression,then the initial value of the object is obtained by evaluating the expression. It is an error if the value of the expression does not belong to the subtype of the object; if the object is an array object, then an implicit subtype conversion is first performed on the value unless the object is a constant whose subtype indication denotes an unconstrained array type. Otherwise, any implicit initial value for the object is determined.
c. The object is created.
d. Any initial value is assigned to the object.

The initialization of such an object (either the declared object or one of its subelements) involves a check that the initial value belongs to the subtype of the object. For an array object declared by an object declaration, an implicit subtype conversion is first applied as for an assignment statement, unless the object is a constant whose subtype is an unconstrained array type.

The elaboration of a file object declaration consists of the elaboration of the subtype indication followed by the creation of the object. If the file object declaration contains file open information, then the implicit call to FILE_OPEN is then executed (see 4.3.1.4).

NOTES

1--These rules apply to all object declarations other than port and generic declarations, which are elaborated as outlined in 12.2.1 through 12.2.4 .

2--The expression initializing a constant object need not be a static expression.

12.3.1.5 Alias declarations

Elaboration of an alias declaration consists of the elaboration of the subtype indication to establish the subtype associated with the alias, followed by the creation of the alias as an alternative name for the named entity. The creation of an alias for an array object involves a check that the subtype associated with the alias includes a matching element for each element of the named object. It is an error if this check fails.

12.3.1.6 Attribute declarations

Elaboration of an attribute declaration has no effect other than to create a template for defining attributes of items.

12.3.1.7 Component declarations

Elaboration of a component declaration has no effect other than to create a template for instantiating component instances.

12.3.2 Elaboration of a specification

Elaboration of a specification has the effect of associating additional information with a previously declared item.

12.3.2.1 Attribute specifications

Elaboration of an attribute specification proceeds as follows:

a. The entity specification is elaborated in order to determine which items are affected by the attribute specification.
b. The expression is evaluated to determine the value of the attribute. It is an error if the value of the expression does not belong to the subtype of the attribute; if the attribute is of an array type, then an implicit subtype conversion is first performed on the value, unless the subtype indication of the attribute denotes an unconstrained array type.
c. A new instance of the designated attribute is created and associated with each of the affected items.
d. Each new attribute instance is assigned the value of the expression.

The assignment of a value to an instance of a given attribute involves a check that the value belongs to the subtype of the designated attribute. For an attribute of a constrained array type, an implicit subtype conversion is first applied as for an assignment statement. No such conversion is necessary for an attribute of an unconstrained array type; the constraints on the value determine the constraints on the attribute.

NOTE--The expression in an attribute specification need not be a static expression.

12.3.2.2 Configuration specifications

Elaboration of a configuration specification proceeds as follows:

a. The component specification is elaborated in order to determine which component instances are affected by the configuration specification.
b. The binding indication is elaborated to identify the design entity to which the affected component instances will be bound.
c. The binding information is associated with each affected component instance label for later use in instantiating those component instances.

As part of this elaboration process, a check is made that both the entity declaration and the corresponding architecture body implied by the binding indication exist within the specified library. It is an error if this check fails.

12.3.2.3 Disconnection specifications

Elaboration of a disconnection specification proceeds as follows:

a. The guarded signal specification is elaborated in order to identify the signals affected by the disconnection specification.
b. The time expression is evaluated to determine the disconnection time for drivers of the affected signals.
c. The disconnection time is associated with each affected signal for later use in constructing disconnection statements in the equivalent processes for guarded assignments to the affected signals.

12.4 Elaboration of a statement part

Concurrent statements appearing in the statement part of a block must be elaborated before execution begins. Elaboration of the statement part of a block consists of the elaboration of each concurrent statement in the order given. This rule holds for all block statement parts except for those blocks equivalent to a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute defined in package STANDARD (see 14.2).

For this case, the statements are not elaborated; instead, the design entity is subject to implementation-dependent elaboration.

12.4.1 Block statements

Elaboration of a block statement consists of the elaboration of the block header, if present, followed by the elaboration of the block declarative part, followed by the elaboration of the block statement part.

Elaboration of a block statement may occur under the control of a configuration declaration. In particular, a block configuration, whether implicit or explicit, within a configuration declaration may supply a sequence of additional implicit configuration specifications to be applied during the elaboration of the corresponding block statement. If a block statement is being elaborated under the control of a configuration declaration, then the sequence of implicit configuration specifications supplied by the block configuration is elaborated as part of the block declarative part, following all other declarative items in that part.

The sequence of implicit configuration specifications supplied by a block configuration, whether implicit or explicit, consists of each of the configuration specifications implied by component configurations (see 1.3.2) occurring immediately within the block configuration, in the order in which the component configurations themselves appear.

12.4.2 Generate statements

Elaboration of a generate statement consists of the replacement of the generate statement with zero or more copies of a block statement whose declarative part consists of the declarative items contained within the generate statement and whose statement part consists of the concurrent statements contained within the generate statement. These block statements are said to be *represented* by the generate statement. Each block statement is then elaborated.

For a generate statement with a for generation scheme, elaboration consists of the elaboration of the discrete range, followed by the generation of one block statement for each value in the range. The block statements all have the following form:

a. The label of the block statement is the same as the label of the generate statement.
b. The block declarative part has, as its first item, a single constant declaration that declares a constant with the same simple name as that of the applicable generate parameter; the value of the constant is the value of the generate parameter for the generation of this particular block statement. The type of this declaration is determined by the base type of the discrete range of the generate parameter. The remainder of the block declarative part consists of a copy of the declarative items contained within the generate statement.
c. The block statement part consists of a copy of the concurrent statements contained within the generate statement.

For a generate statement with an if generation scheme, elaboration consists of the evaluation of the Boolean expression, followed by the generation of exactly one block statement if the expression evaluates to TRUE, and no block statement otherwise. If generated, the block statement has the following form:

-- The block label is the same as the label of the generate statement.

-- The block declarative part consists of a copy of the declarative items contained within the generate statement.

-- The block statement part consists of a copy of the concurrent statements contained within the generate statement.

Examples:

```
--  The following generate statement:

LABL : for I in 1 to 2 generate
    signal s1 : INTEGER;
begin
```

```
        s1 <= p1;
        Inst1 : and_gate port map (s1, p2(I), p3);
    end generate LABL;

    --  is equivalent to the following two block statements:

    LABL : block
        constant I : INTEGER := 1;
        signal s1 : INTEGER;
    begin
        s1 <= p1;
        Inst1 : and_gate port map (s1, p2(I), p3);
    end block LABL;

    LABL : block

        constant I : INTEGER := 2;
        signal s1 : INTEGER;
    begin
        s1 <= p1;
        Inst1 : and_gate port map (s1, p2(I), p3);
    end block LABL;

    --  The following generate statement:

    LABL : if (g1 = g2) generate
        signal s1 : INTEGER;
    begin
        s1 <= p1;
        Inst1 : and_gate port map (s1, p4, p3);
    end generate LABL;

    --  is equivalent to the following statement if g1 = g2;
    --  otherwise, it is equivalent to no statement at all:

    LABL : block
        signal s1 : INTEGER;
    begin
        s1 <= p1;
        Inst1 : and_gate port map (s1, p4, p3);
    end block LABL;
```

NOTE--The repetition of the block labels in the case of a for generation scheme does not produce multiple declarations of the label on the generate statement. The multiple block statements represented by the generate statement constitute multiple references to the same implicitly declared label.

12.4.3 Component instantiation statements

Elaboration of a component instantiation statement that instantiates a component declaration has no effect unless the component instance is either fully bound to a design entity defined by an entity declaration and architecture body or bound to a configuration of such a design entity. If a component instance is so bound, then elaboration of the corresponding component instantiation statement consists of the elaboration of the implied block statement representing the component instance and (within that block) the implied block statement representing the design entity to which the component instance is bound. The implied block statements are defined in 9.6.1 .

Elaboration of a component instantiation statement whose instantiated unit denotes either a design entity or a configuration

declaration consists of the elaboration of the implied block statement representing the component instantiation statement and (within that block) the implied block statement representing the design entity to which the component instance is bound. The implied block statements are defined in 9.6.2 .

12.4.4 Other concurrent statements

All other concurrent statements are either process statements or are statements for which there is an equivalent process statement.

Elaboration of a process statement proceeds as follows:

a. The process declarative part is elaborated.
b. The drivers required by the process statement are created.
c. The initial transaction defined by the default value associated with each scalar signal driven by the process statement is inserted into the corresponding driver.

Elaboration of all concurrent signal assignment statements and concurrent assertion statements consists of the construction of the equivalent process statement followed by the elaboration of the equivalent process statement.

12.5 Dynamic elaboration

The execution of certain constructs that involve sequential statements rather than concurrent statements also involves elaboration. Such elaboration occurs during the execution of the model.

There are three particular instances in which elaboration occurs dynamically during simulation. These are as follows:

a. Execution of a loop statement with a for iteration scheme involves the elaboration of the loop parameter specification prior to the execution of the statements enclosed by the loop (see 8.9). This elaboration creates the loop parameter and evaluates the discrete range.
b. Execution of a subprogram call involves the elaboration of the parameter interface list of the corresponding subprogram declaration; this involves the elaboration of each interface declaration to create the corresponding formal parameters. Actual parameters are then associated with formal parameters. Finally, if the designator of the subprogram is not decorated with the 'FOREIGN attribute defined in package STANDARD, the declarative part of the corresponding subprogram body is elaborated and the sequence of statements in the subprogram body is executed. If the designator of the subprogram is decorated with the 'FOREIGN attribute defined in package STANDARD, then the subprogram body is subject to implementation-dependent elaboration and execution.
c. Evaluation of an allocator that contains a subtype indication involves the elaboration of the subtype indication prior to the allocation of the created object.

NOTE--It is a consequence of these rules that declarative items appearing within the declarative part of a subprogram body are elaborated each time the corresponding subprogram is called; thus, successive elaborations of a given declarative item appearing in such a place may create items with different characteristics. For example, successive elaborations of the same subtype declaration appearing in a subprogram body may create subtypes with different constraints.

12.6 Execution of a model

The elaboration of a design hierarchy produces a *model* that can be executed in order to simulate the design represented by the model. Simulation involves the execution of user-defined processes that interact with each other and with the environment.

The *kernel process* is a conceptual representation of the agent that coordinates the activity of user-defined processes during a simulation. This agent causes the propagation of signal values to occur and causes the values of implicit signals [such as S'Stable(T)] to be updated. Furthermore, this process is responsible for detecting events that occur and for causing the

appropriate processes to execute in response to those events.

For any given signal that is explicitly declared within a model, the kernel process contains a variable representing the current value of that signal. Any evaluation of a name denoting a given signal retrieves the current value of the corresponding variable in the kernel process. During simulation, the kernel process updates that variable from time to time, based upon the current values of sources of the corresponding signal.

In addition, the kernel process contains a variable representing the current value of any implicitly declared GUARD signal resulting from the appearance of a guard expression on a given block statement. Furthermore, the kernel process contains both a driver for, and a variable representing the current value of,any signal S'Stable(T), for any prefix S and any time T, that is referenced within the model; likewise, for any signal S'Quiet(T) or S'Transaction.

12.6.1 Drivers

Every signal assignment statement in a process statement defines a set of *drivers* for certain scalar signals. There is a single driver for a given scalar signal S in a process statement, provided that there is at least one signal assignment statement in that process statement and that the longest static prefix of the target signal of that signal assignment statement denotes S or denotes a composite signal of which S is a subelement. Each such signal assignment statement is said to be *associated* with that driver. Execution of a signal assignment statement affects only the associated driver(s).

A driver for a scalar signal is represented by a *projected output waveform*. A projected output waveform consists of a sequence of one or more *transactions*, where each transaction is a pair consisting of a value component and a time component. For a given transaction, the value component represents a value that the driver of the signal is to assume at some point in time, and the time component specifies which point in time. These transactions are ordered with respect to their time components.

A driver always contains at least one transaction. The initial contents of a driver associated with a given signal are defined by the default value associated with the signal (see 4.3.1.2).

For any driver, there is exactly one transaction whose time component is not greater than the current simulation time. The *current value* of the driver is the value component of this transaction. If, as the result of the advance of time, the current time becomes equal to the time component of the next transaction, then the first transaction is deleted from the projected output waveform and the next becomes the current value of the driver.

12.6.2 Propagation of signal values

As simulation time advances, the transactions in the projected output waveform of a given driver (see 12.6.1) will each, in succession, become the value of the driver. When a driver acquires a new value in this way, regardless of whether the new value is different from the previous value, that driver is said to be *active* during that simulation cycle. For the purposes of defining driver activity, a driver acquiring a value from a null transaction is assumed to have acquired a new value. A signal is said to be *active* during a given simulation cycle

-- If one of its sources is active

-- If one of its subelements is active

-- If the signal is named in the formal part of an association element in a port association list and the corresponding actual is active

-- If the signal is a subelement of a resolved signal and the resolved signal is active

If a signal of a given composite type has a source that is of a different type (and therefore a conversion function or type conversion appears in the corresponding association element), then each scalar subelement of that signal is considered to be active if the source itself is active. Similarly, if a port of a given composite type is associated with a signal that is of a different

type (and therefore a conversion function or type conversion appears in the corresponding association element), then each scalar subelement of that port is considered to be active if the actual signal itself is active.

In addition to the preceding information, an implicit signal is said to be active during a given simulation cycle if the kernel process updates that implicit signal within the given cycle.

If a signal is not active during a given simulation cycle, then the signal is said to be *quiet* during that simulation cycle.

The kernel process determines two values for certain signals during any given simulation cycle. The *driving value* of a given signal is the value that signal provides as a source of other signals. The *effective value* of a given signal is the value obtainable by evaluating a reference to the signal within an expression. The driving value and the effective value of a signal are not always the same, especially when resolution functions and conversion functions or type conversions are involved in the propagation of signal values.

A *basic signal* is a signal that has all of the following properties:

-- It is either a scalar signal or a resolved signal (see 4.3.1.2);

-- It is not a subelement of a resolved signal;

-- Is not an implicit signal of the form S'Stable(T), S'Quiet(T), orS'Transaction (see 14.1); and

-- It is not an implicit signal GUARD (see 9.1).

Basic signals are those that determine the driving values for all other signals.

The driving value of any basic signal S is determined as follows:

-- If S has no source, then the driving value of S is given by the default value associated with S (see 4.3.1.2).

-- If S has one source that is a driver and S is not a resolved signal (see 4.3.1.2), then the driving value of S is the value of that driver.

-- If S has one source that is a port and S is not a resolved signal, then the driving value of S is the driving value of the formal part of the association element that associates S with that port (see 4.3.2.2). The driving value of a formal part is obtained by evaluating the formal part as follows: If no conversion function or type conversion is present in the formal part, then the driving value of the formal part is the driving value of the signal denoted by the formal designator. Otherwise, the driving value of the formal part is the value obtained by applying either the conversion function or type conversion (whichever is contained in the formal part) to the driving value of the signal denoted by the formal designator.

-- If S is a resolved signal and has one or more sources, then the driving values of the sources of S are examined. It is an error if any of these driving values is a composite where one or more subelement values are determined by the null transaction (see 8.4.1) and one or more subelement values are not determined by the null transaction. If S is of signal kind **register** and all the sources of S have values determined by the null transaction, then the driving value of S is unchanged from its previous value. Otherwise, the driving value of S is obtained by executing the resolution function associated with S, where that function is called with an input parameter consisting of the concatenation of the driving values of the sources of S, with the exception of the value of any source of S whose current value is determined by the null transaction.

The driving value of any signal S that is not a basic signal is determined as follows:

-- If S is a subelement of a resolved signal R, the driving value of S is the corresponding subelement value of the

driving value of R.

-- Otherwise (S is a nonresolved, composite signal), the driving value of Sis equal to the aggregate of the driving values of each of the basic signals that are the subelements of S.

For a scalar signal S, the *effective value* of S is determined in the following manner:

-- If S is a signal declared by a signal declaration, a port of mode **buffer**, or an unconnected port of mode **inout**, then the effective value of S is the same as the driving value of S.

-- If S is a connected port of mode **in** or **inout**, then the effective value of S is the same as the effective value of the actual part of the association element that associates an actual with S (see 4.3.2.2). The effective value of an actual part is obtained by evaluating the actual part, using the effective value of the signal denoted by the actual designator in place of the actual designator.

-- If S is an unconnected port of mode **in**, the effective value of S is given by the default value associated with S (see 4.3.1.2).

For a composite signal R, the effective value of R is the aggregate of the effective values of each of the subelements of R.

For a scalar signal S, both the driving and effective values must belong to the subtype of the signal. For a composite signal R, an implicit subtype conversion is performed to the subtype of R; for each element of R, there must be a matching element in both the driving and the resolved value, and vice versa.

In order to update a signal during a given simulation cycle, the kernel process first determines the driving and effective values of that signal. The kernel process then updates the variable containing the current value of the signal with the newly determined effective value, as follows:

a) If S is a signal of some type that is not an array type, the effective value of S is used to update the current value of S. A check is made that the effective value of S belongs to the subtype of S. An error occurs if this subtype check fails. Finally, the effective value of S is assigned to the variable representing the current value of the signal.

b) If S is an array signal (including a slice of an array), the effective value of S is implicitly converted to the subtype of S. The subtype conversion checks that for each element of S there is a matching element in the effective value and vice versa. An error occurs if this check fails. The result of this subtype conversion is then assigned to the variable representing the current value of S.

If updating a signal causes the current value of that signal to change, then an *event* is said to have occurred on the signal. This definition applies to any updating of a signal, whether such updating occurs according to the above rules or according to the rules for updating implicit signals given in 12.6.3 . The occurrence of an event may cause the resumption and subsequent execution of certain processes during the simulation cycle in which the event occurs.

For any signal other than one declared with the signal kind **register**,the driving and effective values of the signal are determined and the current value of that signal is updated as described above in every simulation cycle. A signal declared with the signal kind **register** is updated in the same fashion during every simulation cycle except those in which all of its sources have current values that are determined by null transactions.

A *net* is a collection of drivers, signals (including ports and implicit signals), conversion functions, and resolution functions that, taken together,determine the effective and driving values of every signal on the net.

Implicit signals GUARD S'Stable(T), S'Quiet(T), and S'Transaction, for any prefix S and any time T, are not updated according to the above rules; such signals are updated according to the rules described in 12.6.3 .

NOTES

1--In a simulation cycle, a subelement of a composite signal may be quiet,but the signal itself may be active.

2--The rules concerning association of actuals with formals (see 4.3.2.2) imply that, if a composite signal is associated with a composite port of mode **out**, **inout**, or **buffer**, and if no conversion function or type conversion appears in either the actual or formal part of the association element, then each scalar subelement of the formal is a source of the matching subelement of the actual. In such a case, a given subelement of the actual will be active if and only if the matching subelement of the formal is active.

3--The algorithm for computing the driving value of a scalar signal S is recursive. For example, if S is a local signal appearing as an actual in a port association list whose formal is of mode **out** or **inout**, the driving value of S can only be obtained after the driving value of the corresponding formal part is computed. This computation may involve multiple executions of the above algorithm.

4--Similarly, the algorithm for computing the effective value of a signal S is recursive. For example, if a formal port S of mode **in** corresponds to an actual A, the effective value of A must be computed before the effective value of S can be computed. The actual A may itself appear as a formal port in aport association list.

5--No effective value is specified for **out** and **linkage** ports, since these ports may not be read.

6--Overloading the operator "=" has no effect on the propagation of signal values.

7--A signal of kind **register** may be active even if its associated resolution function does not execute in the current simulation cycle if the values of all of its drivers are determined by the null transaction and at least one of its drivers is also active.

8--The definition of the driving value of a basic signal exhausts all cases, with the exception of a non-resolved signal with more than one source. This condition is defined as an error in 4.3.1.2 .

12.6.3 Updating implicit signals

The kernel process updates the value of each implicit signal GUARD associated with a block statement that has a guard expression. Similarly, the kernel process updates the values of each implicit signal S'Stable(T), S'Quiet(T), or S'Transaction for any prefix S and any time T; this also involves updating the drivers of S'Stable(T) and S'Quiet(T).

For any implicit signal GUARD, the current value of the signal is modified if and only if the corresponding guard expression contains a reference to a signal S and if S is active during the current simulation cycle. In such a case, the implicit signal GUARD is updated by evaluating the corresponding guard expression and assigning the result of that evaluation to the variable representing the current value of the signal.

For any implicit signal S'Stable(T), the current value of the signal (and likewise the current state of the corresponding driver) is modified if and only if one of the following statements is true:

-- An event has occurred on S in this simulation cycle.

-- The driver of S'Stable(T) is active.

If an event has occurred on signal S, then S'Stable(T) is updated by assigning the value FALSE to the variable representing the current value of S'Stable(T), and the driver of S'Stable(T) is assigned the waveform TRUE **after** T. Otherwise, if the driver of S'Stable(T) is active, then S'Stable(T) is updated by assigning the current value of the driver to the variable representing the current value of S'Stable(T). Otherwise, neither the variable nor the driver is modified.

Similarly, for any implicit signal S'Quiet(T), the current value of the signal (and likewise the current state of the corresponding

driver) is modified if and only if one of the following statements is true:

-- S is active.

-- The driver of S'Quiet(T) is active.

If signal S is active, then S'Quiet(T) is updated by assigning the value FALSE to the variable representing the current value of S'Quiet(T), and the driver of S'Quiet(T) is assigned the waveform TRUE **after** T. Otherwise, if the driver of S'Quiet(T) is active, then S'Quiet(T) is updated by assigning the current value of the driver to the variable representing the current value of S'Quiet(T). Otherwise, neither the variable nor the driver is modified.

Finally, for any implicit signal S'Transaction, the current value of the signal is modified if and only if S is active. If signal S is active, then S'Transaction is updated by assigning the value of the expression (**not** S'Transaction) to the variable representing the current value of S'Transaction. At most one such assignment will occur during any given simulation cycle.

For any implicit signal S'Delayed(T), the signal is not updated by the kernel process. Instead, it is updated by constructing an equivalent process (see 14.1) and executing that process.

The current value of a given implicit signal denoted by R is said to *depend* upon the current value of another signal S if one of the following statements is true:

-- R denotes an implicit GUARD signal and S is any other implicit signal named within the guard expression that defines the current value of R.

-- R denotes an implicit signal S'Stable(T).

-- R denotes an implicit signal S'Quiet(T).

-- R denotes an implicit signal S'Transaction.

-- R denotes an implicit signal S'Delayed(T).

These rules define a partial ordering on all signals within a model. The updating of implicit signals by the kernel process is guaranteed to proceed in such a manner that, if a given implicit signal R depends upon the current value of another signal S, then the current value of S will be updated during a particular simulation cycle prior to the updating of the current value of R.

NOTE--These rules imply that, if the driver of S'Stable(T) is active, then the new current value of that driver is the value TRUE. Furthermore, these rules imply that, if an event occurs on S during a given simulation cycle, and if the driver of S'Stable(T) becomes active during the same cycle, the variable representing the current value of S'Stable(T) will be assigned the value FALSE, and the current value of the driver of S'Stable(T) during the given cycle will never be assigned to that signal.

12.6.4 The simulation cycle

The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model. Each such repetition is said to be a *simulation cycle*. In each cycle, the values of all signals in the description are computed. If as a result of this computation an event occurs on a given signal, process statements that are sensitive to that signal will resume and will be executed as part of the simulation cycle.

At the beginning of initialization, the current time, T_C, is assumed to be 0 ns.

The initialization phase consists of the following steps:

-- The driving value and the effective value of each explicitly declared signal are computed, and the current value of the

signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.

-- The value of each implicit signal of the form S'Stable(T) or S'Quiet(T)is set to True. The value of each implicit signal of the form S'Delayed(T) is set to the initial value of its prefix, S.

-- The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard expression.

-- Each nonpostponed process in the model is executed until it suspends.

-- Each postponed process in the model is executed until it suspends.

-- The time of the next simulation cycle (which in this case is the first simulation cycle), T_n, is calculated according to the rules of step f of the simulation cycle, below.

A simulation cycle consists of the following steps:

a. The current time, T_c is set equal to T_n. Simulation is complete when T_n= TIME'HIGH and there are no active drivers or process resumptions at T_n.
b. Each active explicit signal in the model is updated. (Events may occur on signals as a result.)
c. Each implicit signal in the model is updated. (Events may occur on signals as a result.)
d. For each process P, if P is currently sensitive to a signal S and if an event has occurred on S in this simulation cycle, then P resumes.
e. Each nonpostponed process that has resumed in the current simulation cycle is executed until it suspends.
f. The time of the next simulation cycle, T_n, is determined by setting it to the earliest of
    1. TIME'HIGH,
    2. The next time at which a driver becomes active, or
    3. The next time at which a process resumes.
    4. If $T_n = T_c$, then the next simulation cycle (if any) will be a *delta cycle*.
g. If the next simulation cycle will be a delta cycle, the remainder of this step is skipped. Otherwise, each postponed process that has resumed but has not been executed since its last resumption is executed until it suspends. Then T_n is recalculated according to the rules of step f. It is an error if the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle.

NOTES

1--The initial value of any implicit signal of the form S'Transaction is not defined.

2--Updating of explicit signals is described in 12.6.2 ; updating of implicit signals is described in 12.6.3 .

3--When a process resumes, it is added to one of two sets of processes to be executed (the set of postponed processes and the set of nonpostponed processes). However, no process actually begins to execute until all signals have been updated and all executable processes for this simulation cycle have been identified. Nonpostponed processes are always executed during step e of every simulation cycle, while postponed processes are executed during step g of every simulation cycle that does not immediately precede a delta cycle.

4--The second and third steps of the initialization phase and steps b and c of the simulation cycle may occur in interleaved fashion. This interleaving may occur because the implicit signal GUARD may be used as the prefix of another implicit signal; moreover, implicit signals may be associated as actuals with explicit signals, making the value of an explicit signal a function of an implicit signal.

Section 6

Names

The rules applicable to the various forms of name are described in this section.

6.1 Names

Names can denote declared entities, whether declared explicitly or implicitly. Names can also denote

-- Objects denoted by access values,

-- Subelements of composite objects,

-- Subelements of composite values,

-- Slices of composite objects,

-- Slices of composite values, and

-- Attributes of any named entity.

```
name ::=
       simple_name
    | operator_symbol
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name

prefix ::=
       name
    | function_call
```

Certain forms of name (indexed and selected names, slices, and attribute names) include a *prefix* that is a name or a function call. If the prefix of a name is a function call, then the name denotes an element, a slice, or an attribute, either of the result of the function call, or (if the result is an access value) of the object designated by the result. Function calls are defined in 7.3.3 .

If the type of a prefix is an access type, then the prefix must not be a name that denotes a formal parameter of mode **out** or a subelement thereof.

A prefix is said to be *appropriate* for a type in either of the following cases:

-- The type of the prefix is the type considered.

-- The type of the prefix is an access type whose designated type is the type considered.

The evaluation of a name determines the named entity denoted by the name. The evaluation of a name that has a prefix includes the evaluation of the prefix,that is, of the corresponding name or function call. If the type of the prefix is an access type, the evaluation of the prefix includes the determination of the object designated by the corresponding access value. In such a case, it is an error if the value of the prefix is a null access value. It is an error if,after all type analysis (including overload resolution) the name is ambiguous.

A name is said to be a *static name* if and only if one of the following conditions holds:

-- The name is a simple name or selected name (including those that are expanded names) that does not denote a function call or an object or value of an access type and (in the case of a selected name) whose prefix is a static name.

-- The name is an indexed name whose prefix is a static name, and every expression that appears as part of the name is a static expression.

-- The name is a slice name whose prefix is a static name and whose discrete range is a static discrete range.

Furthermore, a name is said to be a *locally static name* if and only if one of the following conditions hold:

-- The name is a simple name or selected name (including those that are expanded names) that is not an alias and that does not denote a function call or an object or a value of an access type and (in the case of a selected name) whose prefix is a locally static name.

-- The name is a simple name or selected name (including those that are expanded names) that is an alias, and that the aliased name given in the corresponding alias declaration (see 4.3.3) is a locally static name, and (in the case of a selected name) whose prefix is a locally static name.

-- The name is an indexed name whose prefix is a locally static name, and every expression that appears as part of the name is a locally static expression.

-- The name is a slice name whose prefix is a locally static name and whose discrete range is a locally static discrete range.

A *static signal name* is a static name that denotes a signal. The *longest static prefix* of a signal name is the name itself, if the name is a static signal name; otherwise, it is the longest prefix of the name that is a static signal name. Similarly, a *static variable name* is a static name that denotes a variable, and the longest static prefix of a variable name is the name itself, if the name is a static variable name; otherwise, it is the longest prefix of the name that is a static variable name.

Examples:

```
S(C,2)            --  A static name: C is a static constant.
R(J to 16)        --  A nonstatic name: J is a signal.
                  --  R is the longest static prefix of R(J to 16).

T(n)              --  A static name; n is a generic constant.
T(2)              --  A locally static name.
```

6.2 Simple names

A simple name for a named entity is either the identifier associated with the entity by its declaration, or another identifier associated with the entity by an alias declaration. In particular, the simple name for an entity interface, a configuration, a package, a procedure, or a function is the identifier that appears in the corresponding entity declaration, configuration declaration, package declaration, procedure declaration, or function declaration, respectively. The simple name of an architecture is that defined by the identifier of the architecture body.

```
simple_name ::=  identifier
```

The evaluation of a simple name has no other effect than to determine the named entity denoted by the name.

6.3 Selected names

A selected name is used to denote a named entity whose declaration appears either within the declaration of another named entity or within a design library.

```
selected_name ::=  prefix . suffix

suffix ::=
       simple_name
     | character_literal
     | operator_symbol
     | all
```

A selected name may be used to denote an element of a record, an object designated by an access value, or a named entity whose declaration is contained within another named entity, particularly within a library or a package. Furthermore, a selected name may be used to denote all named entities whose declarations are contained within a library or a package.

For a selected name that is used to denote a record element, the suffix must be a simple name denoting an element of a record object or value. The prefix must be appropriate for the type of this object or value.

For a selected name that is used to denote the object designated by an access value, the suffix must be the reserved word **all**. The prefix must belong to an access type.

The remaining forms of selected names are called *expanded names*. The prefix of an expanded name may not be a function call.

An expanded name denotes a primary unit contained in a design library if the prefix denotes the library and the suffix is the simple name of a primary unit whose declaration is contained in that library. An expanded name denotes all primary units contained in a library if the prefix denotes the library and the suffix is the reserved word **all**. An expanded name is not allowed for a secondary unit, particularly for an architecture body.

An expanded name denotes a named entity declared in a package if the prefix denotes the package and the suffix is the simple name, character literal, or operator symbol of a named entity whose declaration occurs immediately within that package. An expanded name denotes all named entities declared in a package if the prefix denotes the package and the suffix is the reserved word **all**.

An expanded name denotes a named entity declared immediately within a named construct if the prefix denotes a construct that is an entity interface, an architecture, a subprogram, a block statement, a process statement, a generate statement, or a loop statement, and the suffix is the simple name, character literal, or operator symbol of a named entity whose declaration occurs immediately within that construct. This form of expanded name is only allowed within the construct itself.

If, according to the visibility rules, there is at least one possible interpretation of the prefix of a selected name as the name of an enclosing entity interface, architecture, subprogram, block statement, process statement, generate statement, or loop statement, then the only interpretations considered are those of the immediately preceding paragraph. In this case, the selected name is always interpreted as an expanded name. In particular, no interpretations of the prefix as a function call are considered.

Examples:

```
--  Given the following declarations:
```

```
     type INSTR_TYPE is
        record
            OPCODE:    OPCODE_TYPE;
        end record;
     signal INSTRUCTION: INSTR_TYPE;

     --  The name "INSTRUCTION.OPCODE" is the name of a record element.

     --  Given the following declarations:


     type INSTR_PTR is access INSTR_TYPE;
     variable PTR: INSTR_PTR;

     --  The name "PTR.all" is the name of the object designated by PTR.

     --  Given the following library clause:

     library TTL, CMOS;

     --  The name "TTL.SN74LS221" is the name of a design unit contained in a library
     --  and the name "CMOS.all" denotes all design units contained in a library.

     --  Given the following declaration and use clause:

     library MKS;
     use MKS.MEASUREMENTS, STD.STANDARD;

     --  The name "MEASUREMENTS.VOLTAGE" denotes a named entity declared in a
     --  package and the name "STANDARD.all" denotes all named entities declared in a
     --  package.

     --  Given the following process label and declarative part:


     P: process
         variable DATA: INTEGER;
     begin

     --  Within process P, the name "P.DATA" denotes a named entity declared in
process P.

     end process;
```

NOTES

1--The object denoted by an access value is accessed differently depending on whether the entire object or a subelement of the object is desired. If the entire object is desired, a selected name whose prefix denotes the access value and whose suffix is the reserved word **all** is used. In this case, the access value is not automatically dereferenced, since it is necessary to distinguish an access value from the object denoted by an access value.

If a subelement of the object is desired, a selected name whose prefix denotes the access value is again used; however, the suffix in this case denotes the subelement. In this case, the access value is automatically dereferenced.

These two cases are shown in the following example:

```
     type rec;
```

```
     type recptr is access rec;

     type rec is
          record
                value        : INTEGER;
                \next\       : recptr;
          end record;

     variable list1, list2: recptr;
     variable recobj: rec;

     list2 := list1;          --  Access values are copied;
                                    --  list1 and list2 now denote the same
object.
     list2 := list1.\next\;   --  list2 denotes the same object as list1.\next\.
                                    --  list1.\next\ is the same as
list1.all.\next\.
                                    --  An implicit dereference of the access
value occurs before the
                                    --  "\next\" field is selected.
     recobj := list2.all      --  An explicit dereference is needed here.
```

2--Overload resolution may be used to disambiguate selected names. See rules 1 and 3 of 10.5 .

3--If, according to the rules of this clause and of 10.5 , there is not exactly one interpretation of a selected name that satisfies these rules, then the selected name is ambiguous.

6.4 Indexed names

An indexed name denotes an element of an array.

```
     indexed_name ::=  prefix ( expression { , expression } )
```

The prefix of an indexed name must be appropriate for an array type. The expressions specify the index values for the element; there must be one such expression for each index position of the array, and each expression must be of the type of the corresponding index. For the evaluation of an indexed name,the prefix and the expressions are evaluated. It is an error if an index value does not belong to the range of the corresponding index range of the array.

Examples:

```
     REGISTER_ARRAY(5)          --  An element of a one-dimensional array.
     MEMORY_CELL(1024,7)        --  An element of a two-dimensional array.
```

NOTE--If a name (including one used as a prefix) has an interpretation both as an indexed name and as a function call, then the innermost complete context is used to disambiguate the name. If, after applying this rule, there is not exactly one interpretation of the name, then the name is ambiguous. See 10.5 .

6.5 Slice names

A slice name denotes a one-dimensional array composed of a sequence of consecutive elements of another one-dimensional array. A slice of a signal is a signal; a slice of a variable is a variable; a slice of a constant is a constant; a slice of a value is a value.

```
slice_name ::=  prefix ( discrete_range )
```

The prefix of a slice must be appropriate for a one-dimensional array object. The base type of this array type is the type of the slice.

The bounds of the discrete range define those of the slice and must be of the type of the index of the array. The slice is a *null slice* if the discrete range is a null range. It is an error if the direction of the discrete range is not the same as that of the index range of the array denoted by the prefix of the slice name.

For the evaluation of a name that is a slice, the prefix and the discrete range are evaluated. It is an error if either of the bounds of the discrete range does not belong to the index range of the prefixing array, unless the slice is a null slice. (The bounds of a null slice need not belong to the subtype of the index.)

Examples:

```
signal   R15:  BIT_VECTOR (0 to 31) ;
constant     DATA:       BIT_VECTOR (31 downto 0) ;

R15(0 to 7)                 --  A slice with an ascending range.
DATA(24 downto 1)           --  A slice with a descending range.
DATA(1 downto 24)           --  A null slice.
DATA(24 to 25)              --  An error.
```

NOTE--If A is a one-dimensional array of objects, the name A(N **to** N) or A(N **downto** N) is a slice that contains one element; its type is the base type of A. On the other hand, A(N) is an element of the array A and has the corresponding element type.

6.6 Attribute names

An attribute name denotes a value, function, type, range, signal, or constant associated with a named entity.

```
attribute_name ::=
     prefix [ signature ] ' attribute_designator [ ( expression ) ]

attribute_designator ::=  attribute_simple_name
```

The applicable attribute designators depend on the prefix plus the signature, if any. The meaning of the prefix of an attribute must be determinable independently of the attribute designator and independently of the fact that it is the prefix of an attribute.

A signature may follow the prefix if and only if the prefix denotes a subprogram or enumeration literal, or an alias thereof. In this case, the signature is required to match (see 2.3.2) the parameter and result type profile of exactly one visible subprogram or enumeration literal, as is appropriate to the prefix.

If the attribute designator denotes a predefined attribute, the expression either must or may appear, depending upon the definition of that attribute (see Section 14); otherwise, it must not be present.

If the prefix of an attribute name denotes an alias, then the attribute name denotes an attribute of the aliased name and not the alias itself, except when the attribute designator denotes any of the predefined attributes 'SIMPLE_NAME, 'PATH_NAME, or 'INSTANCE_NAME. If the prefix of an attribute name denotes an alias and the attribute designator denotes any of the predefined attributes SIMPLE_NAME, 'PATH_NAME, or 'INSTANCE_NAME, then the attribute name denotes the attribute of the alias and not of the aliased name.

If the attribute designator denotes a user-defined attribute, the prefix cannot denote a subelement or a slice of an object.

Examples:

```
REG'LEFT(1)              --  The leftmost index bound of array REG.

INPUT_PIN'PATH_NAME      --  The hierarchical path name of the port INPUT_PIN.

CLK'DELAYED(5 ns)        --  The signal CLK delayed by 5 ns.
```

Section 9

Concurrent statements

The various forms of concurrent statements are described in this section. Concurrent statements are used to define interconnected blocks and processes that jointly describe the overall behavior or structure of a design. Concurrent statements execure asynchronously with respect to each other.

```
concurrent_statement ::=
      block_statement
    | process_statement
    | concurrent_procedure_call_statement
    | concurrent_assertion_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | generate_statement
```

The primary concurrent statements are the block statement, which groups together other concurrent statements, and the process statement, which represents a single independent sequential process. Additional concurrent statements provide convenient syntax for representing simple, commonly occurring forms of processes, as well as for representing structural decomposition and regular descriptions.

Within a given simulation cycle, an implementation may execute concurrent statements in parallel or in some order. The language does not define the order, if any, in which such statements will be executed. A description that depends upon a particular order of execution of concurrent statements is erroneous.

All concurrent statements may be labeled. Such labels are implicitly declared at the beginning of the declarative part of the innermost enclosing entity declaration, architecture body, block statement, or generate statement.

9.1 Block statement

A block statement defines an internal block representing a portion of a design. Blocks may be hierarchically nested to support design decomposition.

```
block_statement ::=
     block_label :
          block [ ( guard_expression ) ] [ is ]
              block_header
              block_declarative_part
          begin
              block_statement_part
          end block [ block_label ] ;

block_header ::=
    [ generic_clause
    [ generic_map_aspect ; ] ]
    [ port_clause
    [ port_map_aspect ; ] ]
```

```
block_declarative_part ::=
    { block_declarative_item }

block_statement_part ::=
    { concurrent_statement }
```

If a guard expression appears after the reserved word **block**, then a signal with the simple name GUARD of predefined type BOOLEAN is implicitly declared at the beginning of the declarative part of the block, and the guard expression defines the value of that signal at any given time (see 12.6.4). The type of the guard expression must be type BOOLEAN. Signal GUARD may be used to control the operation of certain statements within the block (see 9.5).

The implicit signal GUARD must not have a source.

If a block header appears in a block statement, it explicitly identifies certain values or signals that are to be imported from the enclosing environment into the block and associated with formal generics or ports. The generic and port clauses define the formal generics and formal ports of the block (see 1.1.1.1 and 1.1.1.2); the generic map and port map aspects define the association of actuals with those formals (see 5.2.1.2). Such actuals are evaluated in the context of the enclosing declarative region.

If a label appears at the end of a block statement, it must repeat the block label.

NOTES

1--The value of signal GUARD is always defined within the scope of a given block, and it does not implicitly extend to design entities bound to components instantiated within the given block. However, the signal GUARD may be explicitly passed as an actual signal in a component instantiation in order to extend its value to lower-level components.

2--An actual appearing in a port association list of a given block can never denote a formal port of the same block.

9.2 Process statement

A process statement defines an independent sequential process representing the behavior of some portion of the design.

```
process_statement ::=
    [ process_label : ]
         [ postponed ] process [ ( sensitivity_list ) ] [ is ]
             process_declarative_part
         begin
            process_statement_part
         end [ postponed ] process [ process_label ] ;

process_declarative_part ::=
    { process_declarative_item }

process_declarative_item ::=
        subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | variable_declaration
      | file_declaration
      | alias_declaration
```

```
        | attribute_declaration
        | attribute_specification
        | use_clause
        | group_type_declaration
        | group_declaration

process_statement_part ::=
    { sequential_statement }
```

If the reserved word **postponed** precedes the initial reserved word **process**, the process statement defines a *postponed process*; otherwise, the process statement defines a *nonpostponed process*.

If a sensitivity list appears following the reserved word **process**, then the process statement is assumed to contain an implicit wait statement as the last statement of the process statement part; this implicit wait statement is of the form

```
wait on sensitivity_list ;
```

where the sensitivity list of the wait statement is that following the reserved word **process**. Such a process statement must not contain an explicit wait statement. Similarly, if such a process statement is a parent of a procedure, then that procedure may not contain a wait statement.

Only static signal names (see 6.1) for which reading is permitted may appear in the sensitivity list of a process statement.

If the reserved word **postponed** appears at the end of a process statement, the process must be a postponed process. If a label appears at the end of a process statement, the label must repeat the process label.

It is an error if a variable declaration in a process declarative part declares a shared variable.

The execution of a process statement consists of the repetitive execution of its sequence of statements. After the last statement in the sequence of statements of a process statement is executed, execution will immediately continue with the first statement in the sequence of statements.

A process statement is said to be a *passive process* if neither the process itself, nor any procedure of which the process is a parent,contains a signal assignment statement. Such a process, or any concurrent statement equivalent to such a process, may appear in the entity statement part of an entity declaration.

NOTES

1--The above rules imply that a process that has an explicit sensitivity list always has exactly one (implicit) wait statement in it, and that wait statement appears at the end of the sequence of statements in the process statement part. Thus, a process with a sensitivity list always waits at the end of its statement part; any event on a signal named in the sensitivity list will cause such a process to execute from the beginning of its statement part down to the end, where it will wait again. Such a process executes once through at the beginning of simulation, suspending for the first time when it executes the implicit wait statement.

2--The time at which a process executes after being resumed by a wait statement(see 8.1) differs depending on whether the process is postponed or nonpostponed. When a nonpostponed process is resumed, it executes in the current simulation cycle (see 2.6.4). When a postponed process is resumed, it does not execute until a simulation cycle occurs in which the next simulation cycle is not a delta cycle. In this way, a postponed process accesses the values of signals that are the "final" values at the current simulated time.

3--The conditions that cause a process to resume execution may no longer hold at the time the process resumes execution if the process is a postponed process.

9.3 Concurrent procedure call statements

A concurrent procedure call statement represents a process containing the corresponding sequential procedure call statement.

```
concurrent_procedure_call_statement ::=
   [ label : ] [ postponed ] procedure_call ;
```

For any concurrent procedure call statement, there is an equivalent process statement. The equivalent process statement is a postponed process if and only if the concurrent procedure call statement includes the reserved word **postponed**. The equivalent process statement has a label if and only if the concurrent procedure call statement has a label; if the equivalent process statement has a label, it is the same as that of the concurrent procedure call statement. The equivalent process statement also has no sensitivity list, an empty declarative part, and a statement part that consists of a procedure call statement followed by a wait statement.

The procedure call statement consists of the same procedure name and actual parameter part that appear in the concurrent procedure call statement.

If there exists a name that denotes a signal in the actual part of any association element in the concurrent procedure call statement, and that actual is associated with a formal parameter of mode **in** or **inout**, then the equivalent process statement includes a final wait statement with a sensitivity clause that is constructed by taking the union of the sets constructed by applying the rule of 8.1 to each actual part associated with a formal parameter.

Execution of a concurrent procedure call statement is equivalent to execution of the equivalent process statement.

Example:

```
     CheckTiming (tPLH, tPHL, Clk, D, Q);                    --  A concurrent
procedure called statement.

     process                                                 --  The equivalent
process.
     begin
         CheckTiming (tPLH, tPHL, Clk, D, Q);
         wait on Clk, D, Q;
     end process;
```

NOTES

1--Concurrent procedure call statements make it possible to declare procedures representing commonly used processes and to create such processes easily by merely calling the procedure as a concurrent statement. The wait statement at the end of the statement part of the equivalent process statement allows a procedure to be called without having it loop interminably, even if the procedure is not necessarily intended for use as a process (i.e., it contains no wait statement). Such a procedure may persist over time (and thus the values of its variables may retain state over time) if its outermost statement is a loop statement and the loop contains a wait statement. Similarly, such a procedure may be guaranteed to execute only once, at the beginning of simulation, if its last statement is a wait statement that has no sensitivity clause, condition clause, or timeout clause.

2--The value of an implicitly declared signal GUARD has no effect on evaluation of a concurrent procedure call unless it is explicitly referenced in one of the actual parts of the actual parameter part of the concurrent procedure call statement.

9.4 Concurrent assertion statements

A concurrent assertion statement represents a passive process statement containing the specified assertion statement.

```
concurrent_assertion_statement ::=
   [ label : ] [ postponed ] assertion ;
```

For any concurrent assertion statement, there is an equivalent process statement. The equivalent process statement is a postponed process if and only if the concurrent assertion statement includes the reserved word **postponed**. The equivalent process statement has a label if and only if the concurrent assertion statement has a label; if the equivalent process statement has a label, it is the same as that of the concurrent assertion statement. The equivalent process statement also has no sensitivity list, an empty declarative part, and a statement part that consists of an assertion statement followed by a wait statement.

The assertion statement consists of the same condition, **report** clause, and **severity** clause that appear in the concurrent assertion statement.

If there exists a name that denotes a signal in the Boolean expression that defines the condition of the assertion, then the equivalent process statement includes a final wait statement with a sensitivity clause that is constructed by applying the rule of 8.1 to that expression; otherwise, the equivalent process statement contains a final wait statement that has no explicit sensitivity clause, condition clause, or timeout clause.

Execution of a concurrent assertion statement is equivalent to execution of the equivalent process statement.

NOTES

1--Since a concurrent assertion statement represents a passive process statement, such a process has no outputs. Therefore, the execution of a concurrent assertion statement will never cause an event to occur. However, if the assertion is false, then the specified error message will be sent to the simulation report.

2--The value of an implicitly declared signal GUARD has no effect on evaluation of the assertion unless it is explicitly referenced in one of the expressions of that assertion.

3--A concurrent assertion statement whose condition is defined by a static expression is equivalent to a process statement that ends in a wait statement that has no sensitivity clause; such a process will execute once through at the beginning of simulation and then wait indefinitely.

9.5 Concurrent signal assignment statements

A concurrent signal assignment statement represents an equivalent process statement that assigns values to signals.

```
concurrent_signal_assignment_statement ::=
     [ label : ] [ postponed ] conditional_signal_assignment
   | [ label : ] [ postponed ] selected_signal_assignment

options ::=  [ guarded ] [ delay_mechanism ]
```

There are two forms of the concurrent signal assignment statement. For each form,the characteristics that distinguish it are discussed in the following paragraphs.

Each form may include one or both of the two options **guarded** and a delay mechanism (see 8.4 for the delay mechanism, 9.5.1 for the conditional signal assignment statement, and 9.5.2 for the selected signal assignment statement).The option **guarded** specifies that the signal assignment statement is executed when a signal GUARD changes from FALSE to TRUE, or when that signal has been TRUE and an event occurs on one of the signal assignment statement's inputs. (The signal GUARD may be one of the implicitly declared GUARD signals associated with block statements that have guard expressions, or it may be an explicitly declared signal of type Boolean that is visible at the point of the concurrent signal assignment statement.) The delay mechanism option specifies the pulse rejection characteristics of the signal assignment statement.

If the target of a concurrent signal assignment is a name that denotes a guarded signal (see 4.3.1.2), or if it is in the form of an aggregate and the expression in each element association of the aggregate is a static signal name denoting a guarded signal, then the target is said to be a *guarded target*. If the target of a concurrent signal assignment is a name that denotes a signal that is not a guarded signal, or if it is in the form of an aggregate and the expression in each element association of the aggregate is a static signal name denoting a signal that is not a guarded signal, then the target is said to be an *unguarded target*. It is an error if the target of a concurrent signal assignment is neither a guarded target nor an unguarded target.

For any concurrent signal assignment statement, there is an equivalent process statement with the same meaning. The process statement equivalent to a concurrent signal assignment statement whose target is a signal name is constructed as follows:

a. If a label appears on the concurrent signal assignment statement, then the same label appears on the process statement.
b. The equivalent process statement is a postponed process if and only if the concurrent signal assignment statement includes the reserved word **postponed**.
c. If the delay mechanism option appears in the concurrent signal assignment, then the same delay mechanism appears in every signal assignment statement in the process statement; otherwise, it appears in no signal assignment statement in the process statement.
d. The statement part of the equivalent process statement consists of a statement transform (described below).

   If the option **guarded** appears in the concurrent signal assignment statement, then the concurrent signal assignment is called a *guarded assignment*. If the concurrent signal assignment statement is a guarded assignment, and if the target of the concurrent signal assignment is a guarded target, then the statement transform is as follows:

```
if GUARD then
   signal_transform
else
   disconnection_statements
end if ;
```

   Otherwise, if the concurrent signal assignment statement is a guarded assignment, but if the target of the concurrent signal assignment is *not*a guarded target, then the statement transform is as follows:

```
if GUARD then
    signal_transform
end if ;
```

   Finally, if the concurrent signal assignment statement is *not* a guarded assignment, and if the target of the concurrent signal assignment is *not* a guarded target, then the statement transform is as follows:

```
signal_transform
```

   It is an error if a concurrent signal assignment is not a guarded assignment and the target of the concurrent signal assignment is a guarded target.

   A *signal transform* is either a sequential signal assignment statement, an if statement, a case statement, or a null statement. If the signal transform is an if statement or a case statement, then it contains either sequential signal assignment statements or null statements, one for each of the alternative waveforms. The signal transform determines which of the alternative waveforms is to be assigned to the output signals.
e. If the concurrent signal assignment statement is a guarded assignment, or if any expression (other than a time expression) within the concurrent signal assignment statement references a signal, then the process statement contains a final wait statement with an explicit sensitivity clause. The sensitivity clause is constructed by taking the union of the sets constructed by applying the rule of 8.1 to each of the aforementioned expressions. Furthermore, if the concurrent signal assignment statement is a guarded assignment, then the sensitivity clause also contains the simple name GUARD. (The signals identified by these names are called the *inputs* of the signal assignment statement.) Otherwise, the process statement contains a final wait statement that has no explicit sensitivity clause, condition clause, or timeout

clause.

Under certain conditions (see above) the equivalent process statement may contain a sequence of disconnection statements. A *disconnection statement* is a sequential signal assignment statement that assigns a null transaction to its target. If a sequence of disconnection statements is present in the equivalent process statement, the sequence consists of one sequential signal assignment for each scalar subelement of the target of the concurrent signal assignment statement. For each such sequential signal assignment, the target of the assignment is the corresponding scalar subelement of the target of the concurrent signal assignment, and the waveform of the assignment is a null waveform element whose time expression is given by the applicable disconnection specification (see 5.3).

If the target of a concurrent signal assignment statement is in the form of an aggregate, then the same transformation applies. Such a target may only contain locally static signal names, and a signal may not be identified by more than one signal name.

It is an error if a null waveform element appears in a waveform of a concurrent signal assignment statement.

Execution of a concurrent signal assignment statement is equivalent to execution of the equivalent process statement.

NOTES

1--A concurrent signal assignment statement whose waveforms and target contain only static expressions is equivalent to a process statement whose final wait statement has no explicit sensitivity clause, so it will execute once through at the beginning of simulation and then suspend permanently.

2--A concurrent signal assignment statement whose waveforms are all the reserved word **unaffected** has no drivers for the target, since every waveform in the concurrent signal assignment statement is transformed to the statement

```
null;
```

in the equivalent process statement. See 9.5.1 .

9.5.1 Conditional signal assignments

The conditional signal assignment represents a process statement in which the signal transform is an if statement.

```
conditional_signal_assignment ::=
   target  <=  options conditional_waveforms ;

conditional_waveforms ::=
     { waveform when condition else }
      waveform [ when condition ]
```

The options for a conditional signal assignment statement are discussed in 9.5 .

For a given conditional signal assignment, there is an equivalent process statement corresponding to it as defined for any concurrent signal assignment statement. If the conditional signal assignment is of the form

```
target <= options     waveform1 when condition1 else
    waveform2 when condition2 else
        .
        .
        .
    waveformN-1 when conditionN-1 else
    waveformN when conditionN;
```

then the signal transform in the corresponding process statement is of the form

```
if condition1 then
     wave_transform1
elsif condition2 then
     wave_transform2
        .
        .
        .
elsif conditionN-1 then
     wave_transformN-1
elsif conditionN then
     wave_transformN
end if ;
```

If the conditional waveform is only a single waveform, the signal transform in the corresponding process statement is of the form

```
wave_transform
```

For any waveform, there is a corresponding *wave transform*. If the waveform is of the form

```
waveform_element1, waveform_element2, ..., waveform_elementN
```

then the wave transform in the corresponding process statement is of the form

```
target <= [ delay_mechanism ] waveform_element1, waveform_element2, ...,
                                    waveform_elementN;
```

If the waveform is of the form

```
unaffected
```

then the wave transform in the corresponding process statement is of the form

```
null;
```

In this example, the final **null** causes the driver to be unchanged, rather than disconnected. (This is the null statement--not a null waveform element).

The characteristics of the waveforms and conditions in the conditional assignment statement must be such that the if statement in the equivalent process statement is a legal statement.

Example:

```
S <= unaffected when Input_pin = S'DrivingValue else
   Input_pin after Buffer_Delay;
```

NOTE--The wave transform of a waveform of the form **unaffected** is the null statement, not the null transaction.

9.5.2 Selected signal assignments

The selected signal assignment represents a process statement in which the signal transform is a case statement.

```
selected_signal_assignment ::=
    with expression select
         target <= options selected_waveforms;

selected_waveforms ::=
    { waveform when choices , }
      waveform when choices
```

The options for a selected signal assignment statement are discussed in 9.5 .

For a given selected signal assignment, there is an equivalent process statement corresponding to it as defined for any concurrent signal assignment statement. If the selected signal assignment is of the form

```
with expression select
    target <= options     waveform1     when choice_list1,
                          waveform2     when choice_list2,
                               .
                               .
                               .
                          waveformN-1   when choice_listN-1,
                          waveformN     when choice_listN ;
```

then the signal transform in the corresponding process statement is of the form

```
case expression is
     when choice_list1 =>
        wave_transform1
     when choice_list2 =>
        wave_transform2
           .
           .
           .
     when choice_listN-1 =>
        wave_transformN-1
     when choice_listN =>
        wave_transformN
end case;
```

Wave transforms are defined in 9.5.1 .

The characteristics of the select expression, the waveforms, and the choices in the selected assignment statement must be such that the case statement in the equivalent process statement is a legal statement.

9.6 Component instantiation statements

A component instantiation statement defines a subcomponent of the design entity in which it appears, associates signals or values with the ports of that subcomponent, and associates values with generics of that subcomponent. This subcomponent is one instance of a class of components defined by a corresponding component declaration, design entity, or configuration declaration.

```
component_instantiation_statement ::=
   instantiation_label :
```

```
            instantiated_unit
                 [ generic_map_aspect ]
                 [ port_map_aspect ] ;

     instantiated_unit ::=
          [ component ] component_name
          | entity entity_name [ ( architecture_identifier ) ]
          | configuration configuration_name
```

The component name, if present, must be the name of a component declared in a component declaration. The entity name, if present, must be the name of a previously analyzed entity interface; if an architecture identifier appears in the instantiated unit, then that identifier must be the same as the simple name of an architecture body associated with the entity declaration denoted by the corresponding entity name. The architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body. The configuration name, if present, must be the name of a previously analyzed configuration declaration. The generic map aspect, if present, optionally associates a single actual with each local generic (or member) in the corresponding component declaration or entity interface. Each local generic member must be associated at most once. Similarly, the port map aspect, if present, optionally associates a single actual with each local port member in the corresponding component declaration or entity interface. Each local port member must be associated at most once. The generic map and port map aspects are described in 5.2.1.2 .

If an instantiated unit containing the reserved word **entity** does not contain an explicitly specified architecture identifier, then the architecture identifier is implicitly specified according to the rules given in 5.2.2 . The architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body.

A component instantiation statement and a corresponding configuration specification, if any, taken together, imply that the block hierarchy within the design entity containing the component instantiation is to be extended with a unique copy of the block defined by another design entity. The generic map and port map aspects in the component instantiation statement and in the binding indication of the configuration specification identify the connections that are to be made in order to accomplish the extension.

NOTES

1--A configuration specification can be used to bind a particular instance of a component to a design entity and to associate the local generics and local ports of the component with the formal generics and formal ports of that design entity. A configuration specification may apply to a component instantiation statement only if the name in the instantiated unit of the component instantiation statement denotes a component declaration. (See 5.2 .)

2--The component instantiation statement may be used to imply a structural organization for a hardware design. By using component declarations, signals,and component instantiation statements, a given (internal or external) block may be described in terms of subcomponents that are interconnected by signals.

3--Component instantiation provides a way of structuring the logical decomposition of a design. The precise structural or behavioral characteristics of a given subcomponent may be described later, provided that the instantiated unit is a component declaration. Component instantiation also provides a mechanism for reusing existing designs in a design library. A configuration specification can bind a given component instance to an existing design entity, even if the generics and ports of the entity declaration do not precisely match those of the component (provided that the instantiated unit is a component declaration); if the generics or ports of the entity declaration do not match those of the component, the configuration specification must contain a generic map or port map, as appropriate, to map the generics and ports of the entity declaration to those of the component.

9.6.1 Instantiation of a component

A component instantiation statement whose instantiated unit contains a name denoting a component is equivalent to a pair of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy

contained in another design unit (i.e., the subcomponent).The outer block represents the component declaration; the inner block represents the design entity to which the component is bound. Each is defined by a block statement.

The header of the block statement corresponding to the component declaration consists of the generic and port clauses (if present) that appear in the component declaration, followed by the generic map and port map aspects (if present) that appear in the corresponding component instantiation statement. The meaning of any identifier appearing in the header of this block statement is associated with the corresponding occurrence of the identifier in the generic clause, port clause, generic map aspect, or port map aspect,respectively. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the design entity.

The header of the block statement corresponding to the design entity consists of the generic and port clauses (if present) that appear in the entity declaration that defines the interface to the design entity, followed by the generic map and port map aspects (if present) that appear in the binding indication that binds the component instance to that design entity. The declarative part of the block statement corresponding to the design entity consists of the declarative items from the entity declarative part, followed by the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the design entity consists of the concurrent statements from the entity statement part, followed by the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the entity declaration or architecture body, respectively.

For example, consider the following component declaration, instantiation, and corresponding configuration specification:

```
component
   COMP port (A,B : inout BIT);
end component;

for C: COMP use
     entity X(Y)
     port map (P1 => A, P2 => B) ;
            .
            .
            .
C:  COMP port map (A => S1, B => S2);
```

Given the following entity declaration and architecture declaration:

```
entity X is
     port (P1, P2 : inout BIT);
     constant Delay: Time := 1 ms;
begin
    CheckTiming (P1, P2, 2*Delay);
end X ;

architecture Y of X is
     signal P3: Bit;
begin
    P3 <= P1 after Delay;
    P2 <= P3 after Delay;
    B:  block
            .
            .
            .
        begin
            .
            .
```

```
                .
            end block;
    end Y;
```

then the following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
    C: block                                                    --  Component block.
            port (A,B : inout BIT);                             --  Local ports.
            port map (A => S1, B => S2);                        --  Actual/local
binding.
        begin
            X:  block                                           --  Design entity block.
                   port (P1, P2 : inout BIT);                   --  Formal ports.
                   port map (P1 => A, P2 => B);                 --  Local/formal
binding.
                   constant Delay: Time := 1 ms;                --  Entity declarative
item.
                   signal P3: Bit;                              --  Architecture
declarative item.
                 begin
               CheckTiming (P1, P2, 2*Delay);                   --  Entity statement.
               P3 <= P1 after Delay;                            --  Architecture
statements.
               P2 <= P3 after Delay;
               B:  block                                        --  Internal block
hierarchy.
                      .
                      .
                      .
               begin
                      .
                      .
                      .
                   end block;
               end block X ;
    end block C;
```

The block hierarchy extensions implied by component instantiation statements that are bound to design entities are accomplished during the elaboration of a design hierarchy (see Section 12).

9.6.2 Instantiation of a design entity

A component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration is equivalent to a pair of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy contained in another design unit (i.e.,the subcomponent). The outer block represents the component instantiation statement; the inner block represents the design entity to which the instance is bound. Each is defined by a block statement.

The header of the block statement corresponding to the component instantiation statement is empty, as is the declarative part of this block statement. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the design entity.

The header of the block statement corresponding to the design entity consists of the generic and port clauses (if present) that appear in the entity declaration that defines the interface to the design entity, followed by the generic map and port map aspects

(if present) that appear in the component instantiation statement that binds the component instance to a copy of that design entity. The declarative part of the block statement corresponding to the design entity consists of the declarative items from the entity declarative part, followed by the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the design entity consists of the concurrent statements from the entity statement part, followed by the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the entity declaration or architecture body, respectively.

For example, consider the following design entity:

```
entity X is
     port (P1, P2: inout BIT);
     constant Delay: DELAY_LENGTH:= 1 ms;
     use WORK.TimingChecks.all;
begin
     CheckTiming(P1, P2, 2*Delay);
end entity X;

architecture Y of X is
     signal P3: BIT;
begin
     P3 <= P1 after Delay;
     P2 <= P3 after Delay;
     B: block
          .
          .
          .
     begin
          .
          .
          .
     end block B;
end architecture Y;
```

This design entity is instantiated by the following component instantiation statement:

```
C: entity Work.X (Y) port map (P1 => S1, P2 => S2);
```

The following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
C: block                                                    --  Instance block.
begin
    X: block                                                --  Design entity block.
        port (P1, P2: inout BIT);                           --  Entity interface
ports.
        port map (P1 => S1, P2 => S2);                      --  Instantiation
statement port map.
        constant Delay: DELAY_LENGTH := 1 ms;               --  Entity declarative
items.
        use WORK.TimingChecks.all;
        signal P3: BIT;                                     --  Architecture
declarative item.
    begin
       CheckTiming (P1, P2, 2*Delay);                       --  Entity statement.
```

```
            P3 <= P1 after Delay;                              --  Architecture
statements.
            P2 <= P3 after Delay;
            B: block
                .
                .
                .
            begin
                .
                .
                .
            end block B;
        end block X;
    end block C;
```

Moreover,consider the following design entity, which is followed by an associated configuration declaration and component instantiation:

```
    entity X is
        port (P1, P2: inout BIT);
        constant Delay: DELAY_LENGTH := 1 ms;
        use WORK.TimingChecks.all;
    begin
        CheckTiming (P1, P2, 2*Delay);
    end entity X;

    architecture Y of X is
        signal P3: BIT;
    begin
        P3 <= P1 after Delay;
        P2 <= P3 after Delay;
        B: block
            .
            .
            .
        begin
            .
            .
            .
        end block B;
    end architecture Y;
```

The configuration declaration is

```
    configuration Alpha of X is
        for Y
            .
            .
            .
        end for;
    end configuration Alpha;
```

The component instantiation is

```
    C: configuration Work.Alpha port map (P1 => S1, P2 => S2);
```

The following block statements implement the coupling between the block hierarchy in which component instantiation statement C appears and the block hierarchy contained in design entity X(Y):

```
     C: block                                                    --  Instance block.
     begin
         X: block                                                --  Design entity
block.
             port (P1, P2: inout BIT);                           --  Entity interface
ports.
             port map (P1 => S1, P2 => S2);                      --  Instantiation
statement port map.
             constant Delay: DELAY_LENGTH := 1 ms;               --  Entity declarative
items.
             use WORK.TimingChecks.all;
             signal P3: BIT;                                     --  Architecture
declarative item.
         begin
            CheckTiming (P1, P2, 2*Delay);                       --  Entity statement.
            P3 <= P1 after Delay;                                --  Architecture
statements.
            P2 <= P3 after Delay;
            B: block
                .
                .
                .
            begin
                .
                .
                .
            end block B;
         end block X;
     end block C;
```

The block hierarchy extensions implied by component instantiation statements that are bound to design entities occur during the elaboration of a design hierarchy(see Section 12).

9.7 Generate statements

A generate statement provides a mechanism for iterative or conditional elaboration of a portion of a description.

```
     generate_statement ::=
         generate_label :
             generation_scheme generate
                 [ { block_declarative_item }
             begin ]
                 { concurrent_statement }
             end generate [ generate_label ] ;

     generation_scheme ::=
          for generate_parameter_specification
       | if condition

     label ::=  identifier
```

If a label appears at the end of a generate statement, it must repeat the generate label.

For a generate statement with a **for** generation scheme, the generate parameter specification is the declaration of the *generate parameter* with the given identifier. The generate parameter is a constant object whose type is the base type of the discrete range of the generate parameter specification.

The discrete range in a generation scheme of the first form must be a static discrete range; similarly, the condition in a generation scheme of the second form must be a static expression.

The elaboration of a generate statement is described in 12.4.2 .

Example:

```
Gen: block
   begin
       L1: CELL port map (Top, Bottom, A(0), B(0)) ;

       L2: for I in 1 to 3 generate
           L3: for J in 1 to 3 generate
               L4: if I+J>4 generate
                   L5: CELL port map (A(I-1),B(J-1),A(I),B(J)) ;
               end generate ;
           end generate ;
       end generate ;
       L6: for I in 1 to 3 generate
           L7: for J in 1 to 3 generate
               L8: if I+J<4 generate
                   L9: CELL port map (A(I+1),B(J+1),A(I),B(J)) ;
               end generate ;
           end generate ;
       end generate ;
   end block Gen;
```

Section 5

Specifications

This section describes *specifications*, which may be used to associate additional information with a VHDL description. A specification associates additional information with a named entity that has been previously declared. There are three kinds of specifications: attribute specifications,configuration specifications, and disconnection specifications.

A specification always relates to named entities that already exist; thus a given specification must either follow or (in certain cases) be contained within the declaration of the entity to which it relates. Furthermore, a specification must always appear either immediately within the same declarative part as that in which the declaration of the named entity appears, or (in the case of specifications that relate to design units or the interface objects of design units, subprograms, or block statements) immediately within the declarative part associated with the declaration of the design unit, subprogram body, or block statement.

5.1 Attribute specification

An attribute specification associates a user-defined attribute with one or more named entities and defines the value of that attribute for those entities. The attribute specification is said to *decorate* the named entity.

```
attribute_specification ::=
     attribute attribute_designator of entity_specification is expression ;

entity_specification ::=
     entity_name_list : entity_class

entity_class ::=
entity          | architecture     | configuration
| procedure     | function         | package
| type          | subtype          | constant
| signal        | variable         | component
| label         | literal          | units
| group         | file

entity_name_list ::=
      entity_designator { , entity_designator }
| others
| all

entity_designator ::=  entity_tag [ signature ]

entity_tag ::=  simple_name | character_literal | operator_symbol
```

The attribute designator must denote an attribute. The entity name list identifies those named entities, both implicitly and explicitly defined, that inherit the attribute, as described below:

-- If a list of entity designators is supplied, then the attribute specification applies to the named entities denoted by those designators. It is an error if the class of those names is not the same as that denoted by the entity class.

-- If the reserved word **others** is supplied, then the attribute specification applies to named entities of the specified class that are declared in the immediately enclosing declarative part, provided that each such entity is not explicitly named in the entity name list of a previous attribute specification for the given attribute.

-- If the reserved word **all** is supplied, then the attribute specification applies to all named entities of the specified class that are declared in the immediately enclosing declarative part.

An attribute specification with the entity name list **others** or **all** for a given entity class that appears in a declarative part must be the last such specification for the given attribute for the given entity class in that declarative part. No named entity in the specified entity class may be declared in a given declarative part following such an attribute specification.

If a name in an entity name list denotes a subprogram or package, it denotes the subprogram declaration or package declaration. Subprogram and package bodies cannot be attributed.

An entity designator that denotes an alias of an object is required to denote the entire object, not a member or subelement (or slice thereof).

The entity tag of an entity designator containing a signature must denote the name of one or more subprograms or enumeration literals. In this case, the signature must match (see 2.3.2) the parameter and result type profile of exactly one subprogram or enumeration literal in the current declarative part; the enclosing attribute specification then decorates that subprogram or enumeration literal.

The expression specifies the value of this attribute for each of the named entities inheriting the attribute as a result of this attribute specification. The type of the expression in the attribute specification must be the same as(or implicitly convertible to) the type mark in the corresponding attribute declaration. If the entity name list denotes an entity interface, architecture body, or configuration declaration, then the expression is required to be locally static (see 7.4).

An attribute specification for an attribute of a design unit (i.e., an entity interface, an architecture, a configuration, or a package) must appear immediately within the declarative part of that design unit. Similarly, an attribute specification for an attribute of an interface object of a design unit, subprogram, or block statement must appear immediately within the declarative part of that design unit, subprogram, or block statement. An attribute specification for an attribute of a procedure, a function, a type, a subtype, an object (i.e., a constant, a file, a signal, or a variable), a component, literal, unit name, group, or a labeled entity must appear within the declarative part in which that procedure, function, type, subtype, object,component, literal, unit name, group, or label, respectively, is explicitly or implicitly declared.

For a given named entity, the value of a user-defined attribute of that entity is the value specified in an attribute specification for that attribute of that entity.

It is an error if a given attribute is associated more than once with a given named entity. Similarly, it is an error if two different attributes with the same simple name (whether predefined or user-defined) are both associated with a given named entity.

An entity designator that is a character literal is used to associate an attribute with one or more character literals. An entity designator that is an operator symbol is used to associate an attribute with one or more overloaded operators.

The decoration of a named entity that can be overloaded attributes all named entities matching the specification already declared in the current declarative part.

If an attribute specification appears, it must follow the declaration of the named entity with which the attribute is associated, and it must precede all references to that attribute of that named entity. Attribute specifications are allowed for all user-defined attributes, but are not allowed for predefined attributes.

An attribute specification may reference a named entity by using an alias for that entity in the entity name list, but such a reference counts as the single attribute specification that is allowed for a given attribute and therefore prohibits a subsequent specification that uses the declared name of the entity (or any other alias) as the entity designator.

An attribute specification whose entity designator contains no signature and identifies an overloaded subprogram has the effect of associating that attribute with each of the designated overloaded subprograms declared within that declarative part.

Examples:

```
attribute PIN_NO of CIN: signal is 10;
attribute PIN_NO of COUT: signal is 5;
attribute LOCATION of ADDER1: label is (10,15);
attribute LOCATION of others: label is (25,77);
attribute CAPACITANCE of all: signal is 15 pF;
attribute IMPLEMENTATION of G1: group is "74LS152";
attribute RISING_DELAY of C2Q: group is 7.2 ns;
```

NOTES

1--User-defined attributes represent local information only and cannot be used to pass information from one description to another. For instance, assume some signal X in an architecture body has some attribute A. Further, assume that X is associated with some local port L of component C. C in turn is associated with some design entity E(B), and L is associated with E's formal port P. Neither L nor P has attributes with the simple name A, unless such attributes are supplied via other attribute specifications; in this latter case, the values of P'A and X'A are not related in any way.

2--The local ports and generics of a component declaration cannot be attributed, since component declarations lack a declarative part.

3--If an attribute specification applies to an overloadable named entity, then declarations of additional named entities with the same simple name are allowed to occur in the current declarative part unless the aforementioned attribute specification has as its entity name list either of the reserved words **others** or **all**.

4--Attribute specifications supplying either of the reserved words **others** or **all** never apply to the interface objects of design units, block statements, or subprograms.

5--An attribute specification supplying either of the reserved words **others** or **all** may apply to none of the named entities in the current declarative part, in the event that none of the named entities in the current declarative part meet all of the requirements of the attribute specification.

5.2 Configuration specification

A configuration specification associates binding information with component labels representing instances of a given component declaration.

```
configuration_specification ::=
   for component_specification binding_indication ;

component_specification ::=
   instantiation_list : component_name

instantiation_list ::=
      instantiation_label { , instantiation_label }
| others
| all
```

The instantiation list identifies those component instances with which binding information is to be associated, as defined below:

-- If a list of instantiation labels is supplied, then the configuration specification applies to the corresponding component instances. Such labels must be (implicitly) declared within the immediately enclosing declarative part. It is an error if these component instances are not instances of the component declaration named in the component specification. It is also an error if any of the labels denote a component instantiation statement whose corresponding instantiated unit does not name a component.

-- If the reserved word **others** is supplied, then the configuration specification applies to instances of the specified component declaration whose labels are (implicitly) declared in the immediately enclosing declarative part,provided that each such component instance is not explicitly named in the instantiation list of a previous configuration specification. This rule applies only to those component instantiation statements whose corresponding instantiated units name components.

-- If the reserved word **all** is supplied, then the configuration specification applies to all instances of the specified component declaration whose labels are (implicitly) declared in the immediately enclosing declarative part. This rule applies only to those component instantiation statements whose corresponding instantiated units name components.

A configuration specification with the instantiation list **others** or **all** for a given component name that appears in a declarative part must be the last such specification for the given component name in that declarative part.

The elaboration of a configuration specification results in the association of binding information with the labels identified by the instantiation list. A label that has binding information associated with it is said to be *bound*. It is an error if the elaboration of a configuration specification results in the association of binding information with a component label that is already bound.

NOTE--A configuration specification supplying either of the reserved words **others** or **all** may apply to none of the component instances in the current declarative part. This is the case when none of the component instances in the current declarative part meet all of the requirements of the given configuration specification.

5.2.1 Binding indication

A binding indication associates instances of a component declaration with a particular design entity. It may also associate actuals with formals declared in the entity interface.

```
binding_indication ::=
    [ use entity_aspect ]
    [ generic_map_aspect ]
    [ port_map_aspect ]
```

The entity aspect of a binding indication, if present, identifies the design entity with which the instances of a component are associated. If present, the generic map aspect of a binding indication identifies the expressions to be associated with formal generics in the design entity interface. Similarly, the port map aspect of a binding indication identifies the signals or values to be associated with formal ports in the design entity interface.

When a binding indication is used in a configuration specification, it is an error if the entity aspect is absent.

A binding indication appearing in a component configuration need not have an entity aspect under the following condition: The block corresponding to the block configuration in which the given component configuration appears is required to have one or more configuration specifications that together configure all component instances denoted in the given component configuration. Under this circumstance, these binding indications are the *primary binding indications*. It is an error if a binding indication appearing in a component configuration does not have an entity aspect and there are no primary binding indications. It is also an error if, under these circumstances, the binding indication has neither a generic map aspect nor a port map aspect. This form of binding indication is the *incremental binding indication*, and it is used to *rebind incrementally* the ports and generics of the denoted instance(s) under the following conditions:

-- For each formal generic appearing in the generic map aspect of the incremental binding indication and denoting a formal generic that is unassociated or associated with **open** in any of the primary binding indications, the given formal generic is bound to the actual with which it is associated in the generic map aspect of the incremental binding indication.

-- For each formal generic appearing in the generic map aspect of the incremental binding indication and denoting a formal generic that is associated with an actual other than **open** in one of the primary binding indications, the given formal generic is *rebound* to the actual with which it is associated in the generic map aspect of the incremental binding indication. That is, the association given in the primary binding indication has no effect for the given instance.

-- For each formal port appearing in the port map aspect of the incremental binding indication and denoting a formal port that is unassociated or associated with **open** in any of the primary binding indications, the given formal port is bound to the actual with which it is associated in the port map aspect of the incremental binding indication.

-- It is an error if a formal port appears in the port map aspect of the incremental binding indication and it is a formal port that is associated with an actual other than **open** in one of the primary binding indications.

If the generic map aspect or port map aspect of a binding indication is not present, then the default rules as described in 5.2.2 apply.

Examples:

```
entity AND_GATE is
    generic (I1toO, I2toO: DELAY_LENGTH := 4 ns);
    port        (I1, I2: in BIT;   O: out BIT);
end entity AND_GATE;

entity XOR_GATE is
    generic (I1toO, I2toO : DELAY_LENGTH := 4 ns);
    port        (I1, I2: in BIT;    O : out BIT);
end entity XOR_GATE;

package MY_GATES is
   component AND_GATE is
      generic  (I1toO, I2toO: DELAY_LENGTH := 4 ns);
      port       (I1, I2: in BIT;   O: out BIT);
   end component AND_GATE;

   component XOR_GATE is
      generic  (I1toO, I2toO: DELAY_LENGTH := 4 ns);
      port       (I1, I2: in BIT;   O : out BIT);
   end component XOR_GATE;
end package MY_GATES;

entity Half_Adder is
    port    (X, Y: in BIT;
                Sum, Carry: out BIT);
end entity Half_Adder;

use WORK.MY_GATES.all;
architecture Structure of Half_Adder is
    for L1: XOR_GATE use
        entity WORK.XOR_GATE(Behavior)             --  The primary binding
indication
            generic map (3 ns, 3 ns)               --  for instance L1.
            port map (I1 => I1, I2 => I2, O => O);
```

```
         for L2: AND_GATE use
             entity WORK.AND_GATE(Behavior)              --  The primary binding
indication
                 generic map (3 ns, 4 ns)                --  for instance L2.
                 port map (I1, open, O);

     begin
         L1: XOR_GATE    port map (X, Y, Sum);
         L2: AND_GATE    port map (X, Y, Carry);
     end architecture Structure;

     use WORK.GLOBAL_SIGNALS.all;
     configuration Different of Half_Adder is
         for Structure
             for L1: XOR_GATE
                 generic map (2.9 ns, 3.6 ns);           --  The  incremental binding
             end for;                                    --  indication of L1;
rebinds its generics.

             for L2: AND_GATE
                 generic map (2.8 ns, 3.25 ns)           --  The incremental binding
                 port map (I2 => Tied_High);             --  indication L2; rebinds
its generics
             end for;                                    --  and binds its open port.
        end for;
     end configuration Different;
```

5.2.1.1 Entity aspect

An entity aspect identifies a particular design entity to be associated with instances of a component. An entity aspect may also specify that such a binding is to be deferred.

```
     entity_aspect ::=
           entity entity_name [ ( architecture_identifier) ]
         | configuration configuration_name
         | open
```

The first form of entity aspect identifies a particular entity declaration and (optionally) a corresponding architecture body. If no architecture identifier appears, then the immediately enclosing binding indication is said to *imply* the design entity whose interface is defined by the entity declaration denoted by the entity name and whose body is defined by the default binding rules for architecture identifiers (see 5.2.2). If an architecture identifier appears, then the immediately enclosing binding indication is said to *imply* the design entity consisting of the entity declaration denoted by the entity name together with an architecture body associated with the entity declaration; the architecture identifier defines a simple name that is used during the elaboration of a design hierarchy to select the appropriate architecture body. In either case, the corresponding component instances are said to be *fully bound*.

At the time of the analysis of an entity aspect of the first form, the library unit corresponding to the entity declaration denoted by the entity name is required to exist; moreover, the design unit containing the entity aspect depends on the denoted entity declaration. If the architecture identifier is also present, the library unit corresponding to the architecture identifier is required to exist only if the binding indication is part of a component configuration containing explicit block configurations or explicit component configurations; only in this case does the design unit containing the entity aspect also depend on the denoted architecture body. In any case, the library unit corresponding to the architecture identifier is required to exist at the time that the design entity implied by the enclosing binding indication is bound to the component instance denoted by the component configuration or configuration specification containing the binding indication; if the library unit corresponding to the

architecture identifier was required to exist during analysis, it is an error if the architecture identifier does not denote the same library unit as that denoted during analysis. The library unit corresponding to the architecture identifier, if it exists, must be an architecture body associated with the entity declaration denoted by the entity name.

The second form of entity aspect identifies a design entity indirectly by identifying a configuration. In this case, the entity aspect is said to *imply* the design entity at the apex of the design hierarchy that is defined by the configuration denoted by the configuration name.

At the time of the analysis of an entity aspect of the second form, the library unit corresponding to the configuration name is required to exist. The design unit containing the entity aspect depends on the configuration denoted by the configuration name.

The third form of entity aspect is used to specify that the identification of the design entity is to be deferred. In this case, the immediately enclosing binding indication is said to *not imply* any design entity. Furthermore,the immediately enclosing binding indication must not include a generic map aspect or a port map aspect.

5.2.1.2 Generic map and port map aspects

A generic map aspect associates values with the formal generics of a block. Similarly, a port map aspect associates signals or values with the formal ports of a block. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements.

```
generic_map_aspect ::=
    generic map ( generic_association_list )

port_map_aspect ::=
    port map ( port_association_list )
```

Both named and positional association are allowed in a port or generic association list.

The following definitions are used in the remainder of this subclause:

-- The term *actual* refers to an actual designator that appears either in an association element of a port association list or in an association element of a generic association list.

-- The term *formal* refers to a formal designator that appears either in an association element of a port association list or in an association element of a generic association list.

The purpose of port and generic map aspects is as follows:

-- Generic map aspects and port map aspects appearing immediately within a binding indication associate actuals with the formals of the design entity interface implied by the immediately enclosing binding indication. No scalar formal may be associated with more than one actual. No scalar subelement of any composite formal may be associated more than once in the same association list.

Each scalar subelement of every local port of the component instances to which an enclosing configuration specification or component configuration applies must be associated as an actual with at least one formal or with a scalar subelement thereof. The actuals of these associations for a given local port may be the entire local port or any slice or subelement (or slice thereof). The actuals in these associations must be locally static names.

-- Generic map aspects and port map aspects appearing immediately within a component instantiation statement associate actuals with the formals of the component instantiated by the statement. No scalar formal may be associated with more than one actual. No scalar subelement of any composite formal may be associated with more than one scalar subelement of an actual.

-- Generic map aspects and port map aspects appearing immediately within a block header associate actuals with the formals defined by the same block header. No scalar formal may be associated with more than one actual. No scalar subelement of any composite formal may be associated with more than one actual or with a scalar subelement thereof.

An actual associated with a formal generic in a generic map aspect must be an expression or the reserved word **open**; an actual associated with a formal port in a port map aspect must be a signal, an expression, or the reserved word **open**.

Certain restrictions apply to the actual associated with a formal port in a port map aspect; these restrictions are described in 1.1.1.2 .

A formal that is not associated with an actual is said to be an *unassociated* formal.

NOTE--A generic map aspect appearing immediately within a binding indication need not associate every formal generic with an actual. These formals may be left unbound so that, for example, a component configuration within a configuration declaration may subsequently bind them.

Example:

```
     entity Buf is
         generic (Buf_Delay: TIME := 0 ns);
         port (Input_pin: in Bit; Output_pin: out Bit);
     end Buf;

     architecture DataFlow of Buf is
     begin
         Output_pin <= Input_pin after Buf_Delay;
     end DataFlow;

     entity Test_Bench is
     end Test_Bench;

     architecture Structure of Test_Bench is
         component Buf is
            generic (Comp_Buf_Delay: TIME);
            port (Comp_I: in Bit; Comp_O: out Bit);
         end component;

         --  A binding indication; generic and port map aspects within a binding
indication
         --  associate actuals (Comp_I, etc.) with formals of the design entity
interface
         --  (Input_pin, etc.):
         for UUT: Buf
              use entity Work.Buf(DataFlow)
                   generic map (Buf_Delay => Comp_Buf_Delay)
                   port map (Input_pin => Comp_I, Output_pin=> Comp_O);
         signal S1,S2: Bit;
     begin

         --  A component instantiation statement; generic and port map aspects within
a
         --  component instantiation statement associate actuals (S1, etc.) with the
         --  formals of a component (Comp_I, etc.):
         UUT: Buf
            generic map(Comp_Buf_Delay => 50 ns)
            port map(Comp_I => S1, Comp_O => S2);
```

```
        --  A block statement; generic and port map aspects within the block header
of a block
        --  statement associate actuals (4, etc.) with the formals defined in the
block header:
        B: block
           generic (G: INTEGER);
           generic map(G => 4);
        begin
        end block;
    end Structure;
```

NOTE--A local generic (from a component declaration) or formal generic (from a block statement or from the entity declaration of the enclosing design entity) may appear as an actual in a generic map aspect. Similarly, a local port (from a component declaration) or formal port (from a block statement or from the entity declaration of the enclosing design entity) may appear as an actual in a port map aspect.

5.2.2 Default binding indication

In certain circumstances, a default binding indication will apply in the absence of an explicit binding indication. The default binding indication consists of a default entity aspect, together with a default generic map aspect and a default port map aspect, as appropriate.

If no visible entity declaration has the same simple name as that of the instantiated component, then the default entity aspect is **open**. A *visible entity declaration* is either

a) An entity declaration that has the same simple name as that of the instantiated component and that is directly visible (see 10.3), or

b) An entity declaration that has the same simple name as that of the instantiated component and that would be directly visible in the absence of a directly visible (see 10.3) component declaration with the same simple name as that of the entity declaration

These visibility checks are made at the point of the absent explicit binding indication that causes the default binding indication to apply.

Otherwise, the default entity aspect is of the form

```
entity entity_name ( architecture_identifier )
```

where the entity name is the simple name of the instantiated component, and the architecture identifier is the same as the simple name of the most recently analyzed architecture body associated with the entity declaration. If this rule is applied either to a binding indication contained within a configuration specification or to a component configuration that does not contain an explicit inner block configuration, then the architecture identifier is determined during elaboration of the design hierarchy containing the binding indication. Likewise, if a component instantiation statement contains an instantiated unit containing the reserved word **entity** but does not contain an explicitly specified architecture identifier, this rule is applied during the elaboration of the design hierarchy containing a component instantiation statement. In all other cases, this rule is applied during analysis of the binding indication.

It is an error if there is no architecture body associated with the entity interface denoted by an entity name that is the simple name of the instantiated component.

The default binding indication includes a default generic map aspect if the design entity implied by the entity aspect contains formal generics. The default generic map aspect associates each local generic in thecorresponding component instantiation (if

any) with a formal of the same simple name. It is an error if such a formal does not exist or if its mode and type are not appropriate for such an association. Any remaining unassociated formals are associated with the actual designator **open**.

The default binding indication includes a default port map aspect if the design entity implied by the entity aspect contains formal ports. The default port map aspect associates each local port in the corresponding component instantiation (if any) with a formal of the same simple name. It is an error if such a formal does not exist or if its mode and type are not appropriate for such an association. Any remaining unassociated formals are associated with the actual designator **open**.

If an explicit binding indication lacks a generic map aspect, and if the design entity implied by the entity aspect contains formal generics, then the default generic map aspect is assumed within that binding indication. Similarly, if an explicit binding indication lacks a port map aspect, and the design entity implied by the entity aspect contains formal ports, then the default port map aspect is assumed within that binding indication.

5.3 Disconnection specification

A disconnection specification defines the time delay to be used in the implicit disconnection of drivers of a guarded signal within a guarded signal assignment.

```
disconnection_specification ::=
    disconnect guarded_signal_specification after time_expression ;

guarded_signal_specification ::=
    guarded_signal_list : type_mark

signal_list ::=
        signal_name { , signal_name }
  | others
  | all
```

Each signal name in a signal list in a guarded signal specification must be a locally static name that denotes a guarded signal (see 4.3.1.2). Each guarded signal must be an explicitly declared signal or member of such a signal.

If the guarded signal is a declared signal or a slice thereof, the type mark must be the same as the type mark indicated in the guarded signal specification (see 4.3.1.2). If the guarded signal is an array element of an explicitly declared signal, the type mark must be the same as the element subtype indication in the (explicit or implicit) array type declaration that declares the base type of the explicitly declared signal. If the guarded signal is a record element of an explicitly declared signal, then the type mark must be the same as the type mark in the element subtype definition of the record type declaration that declares the type of the explicitly declared signal. Each signal must be declared in the declarative part enclosing the disconnection specification.

Subject to these rules, a disconnection specification *applies to* the drivers of a guarded signal S of whose type mark denotes the type T under the following circumstances:

-- For a scalar signal S, if an explicit or implicit disconnection specification of the form

```
disconnect S: T after time_expression;
```

exists, then this disconnection specification applies to the drivers of S.

-- For a composite signal S, an explicit or implicit disconnection specification of the form

```
disconnect S: T after time_expression;
```

is equivalent to a series of implicit disconnection specifications, one for each scalar subelement of the signal S. Each disconnection specification in the series is created as follows: it has, as its single signal name in its signal list, a unique scalar subelement of S. Its type mark is the same as the type of the same scalar subelement of S. Its time expression is the same as that of the original disconnection specification.

The characteristics of the disconnection specification must be such that each implicit disconnection specification in the series is a legal disconnection specification.

-- If the signal list in an explicit or implicit disconnection specification contains more than one signal name, the disconnection specification is equivalent to a series of disconnection specifications, one for each signal name in the signal list. Each disconnection specification in the series is created as follows: It has, as its single signal name in its signal list, a unique member of the signal list from the original disconnection specification. Its type mark and time expression are the same as those in the original disconnection specification.

The characteristics of the disconnection specification must be such that each implicit disconnection specification in the series is a legal disconnection specification.

-- An explicit disconnection specification of the form

```
disconnect others: T after time_expression;
```

is equivalent to an implicit disconnection specification where the reserved word **others** is replaced with a signal list comprised of the simple names of those guarded signals that are declared signals declared in the enclosing declarative part, whose type mark is the same as T, and that do not otherwise have an explicit disconnection specification applicable to its drivers; the remainder of the disconnection specification is otherwise unchanged. If there are no guarded signals in the enclosing declarative part whose type mark is the same as T and that do not otherwise have an explicit disconnection specification applicable to its drivers, then the above disconnection specification has no effect.

The characteristics of the explicit disconnection specification must be such that the implicit disconnection specification, if any, is a legal disconnection specification.

-- An explicit disconnection specification of the form

```
disconnect all: T after time_expression;
```

is equivalent to an implicit disconnection specification where the reserved word **all** is replaced with a signal list comprised of the simple names of those guarded signals that are declared signals declared in the enclosing declarative part and whose type mark is the same as T; the remainder of the disconnection specification is otherwise unchanged. If there are no guarded signals in the enclosing declarative part whose type mark is the same as T,then the above disconnection specification has no effect.

The characteristics of the explicit disconnection specification must be such that the implicit disconnection specification, if any, is a legal disconnection specification.

A disconnection specification with the signal list **others** or **all** for a given type that appears in a declarative part must be the last such specification for the given type in that declarative part. No guarded signal of the given type may be declared in a given declarative part following such a disconnection specification.

The time expression in a disconnection specification must be static and must evaluate to a non-negative value.

It is an error if more than one disconnection specification applies to drivers of the same signal.

If, by these rules, no disconnection specification applies to the drivers of a guarded, scalar signal S whose type mark is T (including a scalar subelement of a composite signal), then the following default disconnection specification is implicitly

assumed:

```
disconnect S : T after 0 ns;
```

A disconnection specification that applies to the drivers of a guarded signal S is the *applicable disconnection specification* for the signal S.

Thus the implicit disconnection delay for any guarded signal is always defined,either by an explicit disconnection specification or by an implicit one.

NOTES

1--A disconnection specification supplying either the reserved words **others** or **all** may apply to none of the guarded signals in the current declarative part, in the event that none of the guarded signals in the current declarative part meet all of the requirements of the disconnection specification.

2--Since disconnection specifications are based on declarative parts, not on declarative regions, ports declared in an entity interface cannot be referenced by a disconnection specification in a corresponding architecture body.

Cross-References: Disconnection statements, 9.5 ; Guarded assignment, 9.5 ; Guarded blocks, 9.1 ; Guarded signals, 4.3.1.2 ; Guarded targets, 9.5 ; Signal guard, 9.1 .

Section 11

Design units and their analysis

The overall organization of descriptions, as well as their analysis and subsequent definition in a design library, are discussed in this section.

11.1 Design units

Certain constructs may be independently analyzed and inserted into a design library; these constructs are called *design units*. One or more design units in sequence comprise a *design file*.

```
design_file ::=  design_unit { design_unit }

design_unit ::=  context_clause library_unit

library_unit ::=
       primary_unit
    | secondary_unit

primary_unit ::=
       entity_declaration
    | configuration_declaration
    | package_declaration

secondary_unit ::=
       architecture_body
    | package_body
```

Design units in a design file are analyzed in the textual order of their appearance in the design file. Analysis of a design unit defines the corresponding library unit in a design library. A *library unit* is either a primary unit or a secondary unit. A secondary unit is a separately analyzed body of a primary unit resulting from a previous analysis.

The name of a primary unit is given by the first identifier after the initial reserved word of that

unit. Of the secondary units, only architecture bodies are named; the name of an architecture body is given by the identifier following the reserved word **architecture**. Each primary unit in a given library must have a simple name that is unique within the given library, and each architecture body associated with a given entity declaration must have a simple name that is unique within the set of names of the architecture bodies associated with that entity declaration.

Entity declarations, architecture bodies, and configuration declarations are discussed in Section 1. Package declarations and package bodies are discussed in Section 2.

11.2 Design libraries

A *design library* is an implementation-dependent storage facility for previously analyzed design units. A given implementation is required to support any number of design libraries.

```
library_clause  ::=  library logical_name_list ;

logical_name_list  ::=  logical_name { , logical_name }

logical_name  ::=  identifier
```

A library clause defines logical names for design libraries in the host environment. A library clause appears as part of a context clause at the beginning of a design unit. There is a certain region of text called the *scope* of a library clause; this region starts immediately after the library clause, and it extends to the end of the declarative region associated with the design unit in which the library clause appears. Within this scope each logical name defined by the library clause is directly visible, except where hidden in an inner declarative region by a homograph of the logical name according to the rules of 10.3 .

If two or more logical names having the same identifier (see 13.3) appear in library clauses in the same context clause, the second and subsequent occurrences of the logical name have no effect. The same is true of logical names appearing both in the context clause of a primary unit and in the context clause of a corresponding secondary unit.

Each logical name defined by the library clause denotes a design library in the host environment.

For a given library logical name, the actual name of the corresponding design libraries in the host environment may or may not be the same. A given implementation must provide some mechanism to associate a library logical name with a host-dependent library. Such a mechanism is not defined by the language.

There are two classes of design libraries: working libraries and resource libraries. A *working library* is the library into which the library unit resulting from the analysis of a design unit is placed. A *resource library* is a library containing library units that are referenced within the design unit being analyzed. Only one library may be the working library during the analysis of any given design unit; in contrast, any number of libraries (including the working library itself) may be resource libraries during such an analysis.

Every design unit except package STANDARD is assumed to contain the following implicit context items as part of its context clause:

```
library STD, WORK ; use STD.STANDARD.all ;
```

Library logical name STD denotes the design library in which package STANDARD and package TEXTIO reside; these are the only standard packages defined by the language (see Section 14). (The use clause makes all declarations within package STANDARD directly visible within the corresponding design unit; see 10.4). Library logical name WORK denotes the current working library during a given analysis.

The library denoted by the library logical name STD contains no library units other than package STANDARD and package TEXTIO.

A secondary unit corresponding to a given primary unit may only be placed into the design library in which the primary unit resides.

NOTE--The design of the language assumes that the contents of resource libraries named in all library clauses in the context clause of a design unit will remain unchanged during the analysis of that unit (with the possible exception of the updating of the library unit corresponding to the analyzed design unit within the working library, if that library is also a resource library).

11.3 Context clauses

A context clause defines the initial name environment in which a design unit is analyzed.

```
context_clause ::=  { context_item }

context_item  ::=
      library_clause
    | use_clause
```

A library clause defines library logical names that may be referenced in the design unit; library

clauses are described in 11.2 . A use clause makes certain declarations directly visible within the design unit; use clauses are described in 10.4 .

NOTE--The rules given for use clauses are such that the same effect is obtained whether the name of a library unit is mentioned once or more than once by the applicable use clauses, or even within a given use clause.

11.4 Order of analysis

The rules defining the order in which design units can be analyzed are direct consequences of the visibility rules. In particular:

a. A primary unit whose name is referenced within a given design unit must be analyzed prior to the analysis of the given design unit.
b. A primary unit must be analyzed prior to the analysis of any corresponding secondary unit.

In each case, the second unit *depends on* the first unit.

The order in which design units are analyzed must be consistent with the partial ordering defined by the above rules.

If any error is detected while attempting to analyze a design unit, then the attempted analysis is rejected and has no effect whatsoever on the current working library.

A given library unit is potentially affected by a change in any library unit whose name is referenced within the given library unit. A secondary unit is potentially affected by a change in its corresponding primary unit. If a library unit is changed (e.g., by reanalysis of the corresponding design unit),then all library units that are potentially affected by such a change become obsolete and must be reanalyzed before they can be used again.

Section 13

Lexical elements

The text of a description consists of one or more design files. The text of a design file is a sequence of lexical elements, each composed of characters; the rules of composition are given in this section.

13.1 Character set

The only characters allowed in the text of a VHDL description are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the ISO eight-bit coded character set [(ISO 8859-1 : 1987 (E)], and is represented (visually) by a graphical symbol.

```
basic_graphic_character ::=
     upper_case_letter | digit | special_character |space_character

graphic_character ::=
     basic_graphic_character | lower_case_letter | other_special_character

basic_character ::=
     basic_graphic_character | format_effector
```

The basic character set is sufficient for writing any description. The characters included in each of the categories of basic graphic characters are defined as follows:

a. Uppercase letters
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ
b. Digits
   0 1 2 3 4 5 6 7 8 9
c. Special characters
   " # & ' ( ) * + , - . / : ; < = > [ ] _ |
d. The space characters
   SPACE[1], NBSP[2]

Format effectors are the ISO (and ASCII) characters called horizontal tabulation,vertical tabulation, carriage return, line feed, and form feed.

[1]The visual representation of the space is the absence of a graphic symbol. It may be interpreted as a graphic character, a control character, or both.

[2]The visual representation of the nonbreaking space is the absence of a graphic symbol. It is used when a line break is to be prevented in the text as presented.

The characters included in each of the remaining categories of graphic characters are defined as follows:

a. Lowercase letters
a b c d e f g h i j k l m n o p q r s t u v w x y z ß à á â ã ä å æ ç è é ê ë ì
í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ

b. Other special characters
! $ % @ ? \ ^ ` { } ~ ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ~ ® ¯ ° ± ² ³ ' µ ¶ · ¸ ¹ º » 1/4
1/2 3/4 ¿ × ÷ -

NOTES

Allowable replacements for the special characters vertical line(|), number sign (#), and quotation mark (") are defined in the last clause of this section.

1--The font design of graphical symbols (for example, whether they are in italic or bold typeface) is not part of ISO 8859-1:1987.

2--The meanings of the acronyms used in this section are as follows: ASCII stands for American Standard Code for Information Interchange, ISO stands for International Organization for Standardization.

3--There are no uppercase equivalents for the characters ß and ÿ.

4--The following names are used when referring to special characters:

Character	Name	Character	Name
"	quotation mark	¢	cent sign
#	number sign	£	pound sign
&	ampersand	¤	currency sign
'	apostrophe, tick	¥	yen sign
(	left parenthesis	¦	broken bar
)	right parenthesis	§	paragraph sign, section sign
*	asterisk, multiply	¨	diaeresis
+	plus sign	©	copyright sign
,	comma	ª	feminine ordinal indicator
-	hyphen, minus sign	«	left angle quotation mark
.	dot, point, period, full stop	-	not sign
/	slash, divide, solidus	-	soft hyphen*
:	colon	®	registered trade mark sign
;	semicolon	¯	macron
<	less-than sign	°	ring above, degree sign
=	equals sign	±	plus-minus sign
>	greater-than sign	²	superscript two
_	underline, low line	³	superscript three
|	vertical line, vertical bar	'	acute accent
!	exclamation mark	µ	micro sign
$	dollar sign	¶	pilcrow sign
%	percent sign	·	middle dot
?	question mark	¸	cedilla

@	commercial at	¹	superscript one
[	left square bracket	º	masculine ordinal indicator
\	backslash, reverse solidus	»	right angle quotation mark
]	right square bracket	¼	vulgar fraction one quarter
^	circumflex accent	½	vulgar fraction one half
`	grave accent	¾	vulgar fraction three quarters
{	left curly bracket	¿	inverted question mark
}	right curly bracket	×	multiplication sign
~	tilde	÷	division sign
¡	inverted exclamation mark		

*The soft hyphen is a graphic character that is imaged by a graphic symbol identical with, or similar to, that representing HYPHEN, for use when a line break has been established within a work.

13.2 Lexical elements, separators, and delimiters

The text of each design unit is a sequence of separate lexical elements. Each lexical element is either a delimiter, an identifier (which may be a reserved word), an abstract literal, a character literal, a string literal, a bit string literal, or a comment.

In some cases an explicit separator is required to separate adjacent lexical elements (namely when, without separation, interpretation as a single lexical element is possible). A separator is either a space character (SPACE or NBSP),a format effector, or the end of a line. A space character (SPACE or NBSP) is a separator except within a comment, a string literal, or a space character literal.

The end of a line is always a separator. The language does not define what causes the end of a line. However if, for a given implementation, the end of aline is signified by one or more characters, then these characters must be format effectors other than horizontal tabulation. In any case, a sequence of one or more format effectors other than horizontal tabulation must cause atleast one end-of-line.

One or more separators are allowed between any two adjacent lexical elements,before the first of each design unit or after the last. At least one separator is required between an identifier or an abstract literal and an adjacent identifier or abstract literal.

A delimiter is either one of the following special characters (in the basic character set):

```
& ' ( ) * + , - . / : ; < = > | [ ]
```

or one of the following compound delimiters, each composed of two adjacent special characters:

```
=> ** := /= >= <= <>
```

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter or

as a character of a comment, string literal, character literal, or abstract literal.

The remaining forms of lexical elements are described in other clauses of this section.

NOTES

1--Each lexical element must fit on one line, since the end of a line is a separator. The quotation mark, number sign, and underline characters, likewise two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

2--The following names are used when referring to compound delimiters:

Delimiter	Name
=>	arrow
**	double star, exponentiate
:=	variable assignment
/=	inequality (pronounced "not equal")
>=	greater than or equal
<=	less than or equal; signal assignment
<>	box

13.3 Identifiers

Identifiers are used as names and also as reserved words.

```
identifier ::=  basic_identifier | extended_identifier
```

13.3.1 Basic identifiers

A basic identifier consists only of letters, digits, and underlines.

```
basic_identifier ::=
    letter  { [ underline ] letter_or_digit }

letter_or_digit ::=  letter | digit

letter ::=  upper_case_letter | lower_case_letter
```

All characters of a basic identifier are significant, including any underline character inserted between a letter or digit and an adjacent letter or digit. Basic identifiers differing only in the use of corresponding upper and lowercase letters are considered the same.

Examples:

```
COUNT    X     c_out      FFT     Decoder

VHSIC    X1    PageCount  STORE_NEXT_ITEM
```

NOTE--No space (SPACE or NBSP) is allowed within a basic identifier since a space is a separator.

13.3.2 Extended identifiers

Extended identifiers may contain any graphic character.

```
extended_identifier ::=
   \ graphic_character { graphic_character } \
```

If a backslash is to be used as one of the graphic characters of an extended literal, it must be doubled. All characters of an extended identifier are significant (a doubled backslash counting as one character). Extended identifiers differing only in the use of corresponding upper and lowercase letters are distinct. Moreover, every extended identifier is distinct from any basic identifier.

Examples:

```
\BUS\     \bus\                          --  Two different identifiers, neither of which is
                                         --  the reserved word bus.

\a\\b\                                   --  An identifier containing three characters.

VHDL     \VHDL\     \vhdl\               --  Three distinct identifiers.
```

13.4 Abstract literals

There are two classes of abstract literals: real literals and integer literals. A real literal is an abstract literal that includes a point; an integer literal is an abstract literal without a point. Real literals are the literals of the type *universal_real*. Integer literals are the literals of the type *universal_integer*.

```
abstract_literal ::=  decimal_literal | based_literal
```

13.4.1 Decimal literals

A decimal literal is an abstract literal expressed in the conventional decimal notation (that is, the base is implicitly ten).

```
decimal_literal ::=  integer [ . integer ] [ exponent ]

integer ::=  digit  { [ underline ] digit }

exponent ::=  E [ + ] integer | E - integer
```

An underline character inserted between adjacent digits of a decimal literal does not affect the value of this abstract literal. The letter E of the exponent,if any, can be written either in lowercase or in uppercase, with the same meaning.

An exponent indicates the power of ten by which the value of the decimal literal without the exponent is to be multiplied to obtain the value of the decimal literal

with the exponent. An exponent for an integer literal must not have a minus sign.

Examples:

```
     12          0          1E6          123_456               --  Integer literals

     12.0        0.0        0.456        3.14159_26            --  Real literals

     1.34E-12    1.0E+6     6.023E+24                          --  Real literals with
exponents
```

13.4.2 Based literals

A based literal is an abstract literal expressed in a form that specifies the base explicitly. The base must be at least two and at most sixteen.

```
     based_literal ::=
         base # based_integer [ . based_integer ] # [ exponent ]

     base ::=  integer

     based_integer ::=
         extended_digit { [ underline ] extended_digit }

     extended_digit ::=  digit | letter
```

An underline character inserted between adjacent digits of a based literal does not affect the value of this abstract literal. The base and the exponent, if any, are in decimal notation. The only letters allowed as extended digits are the letters A through F for the digits ten through fifteen. A letter in a based literal (either an extended digit or the letter E of an exponent) can be written either in lowercase or in uppercase, with the same meaning.

The conventional meaning of based notation is assumed; in particular the value of each extended digit of a based literal must be less than the base. An exponent indicates the power of the base by which the value of the based literal without the exponent is to be multiplied to obtain the value of the based literal with the exponent. An exponent for a based integer literal must not have a minus sign.

Examples:

```
     --  Integer literals of value 255:
          2#1111_1111#               16#FF#             016#0FF#
     --  Integer literals of value 224:
          16#E#E1                    2#1110_0000#
     --  Real literals of value   4095.0 :
          16#F.FF#E+2                2#1.1111_1111_111#E11
```

13.5 Character literals

A character literal is formed by enclosing one of the 191 graphic characters(including the space and nonbreaking space characters) between two

apostrophe characters. A character literal has a value that belongs to a character type.

```
character_literal ::= ' graphic_character '
```

Examples:

```
'A'   '*'   '''   ' '
```

13.6 String literals

A string literal is formed by a sequence of graphic characters (possibly none)enclosed between two quotation marks used as string brackets.

```
string_literal ::=  " { graphic_character } "
```

A string literal has a value that is a sequence of character values corresponding to the graphic characters of the string literal apart from the quotation mark itself. If a quotation-mark value is to be represented in the sequence of character values, then a pair of adjacent quotation marks must be written at the corresponding place within the string literal. (This means that a string literal that includes two adjacent quotation marks is never interpreted as two adjacent string literals.)

The length of a string literal is the number of character values in the sequence represented. (Each doubled quotation mark is counted as a single character.)

Examples:

```
"Setup time is too short"               --  An error message.

""                                      --  An empty string literal.

" "       "A"      """"                 --  Three string literals of length 1.

"Characters such as $, %, and } are allowed in string literals."
```

NOTE--A string literal must fit on one line, since it is a lexical element (see 13.2).Longer sequences of graphic character values can be obtained by concatenation of string literals. The concatenation operation may also be used to obtain string literals containing nongraphic character values. The predefined type CHARACTER in package STANDARD specifies the enumeration literals denoting both graphic and nongraphic characters. Examples of such uses of concatenation are

```
"FIRST PART OF A SEQUENCE OF CHARACTERS " &
"THAT CONTINUES ON THE NEXT LINE"

"Sequence that includes the" & ACK & "control character"
```

13.7 Bit string literals

A bit string literal is formed by a sequence of extended digits (possibly none)enclosed between two quotations used as bit string brackets, preceded by a base

specifier.

bit_string_literal ::= base_specifier " [bit_value] "

bit_value ::= extended_digit { [underline] extended_digit }

base_specifier ::= B | O | X

An underline character inserted between adjacent digits of a bit string literal does not affect the value of this literal. The only letters allowed as extended digits are the letters A through F for the digits ten through fifteen. A letter in a bit string literal (either an extended digit or the base specifier) can be written either in lowercase or in uppercase, with the same meaning.

If the base specifier is 'B', the extended digits in the bit value are restricted to 0 and 1. If the base specifier is 'O', the extended digits in the bit value are restricted to legal digits in the octal number system, i.e.,the digits 0 through 7. If the base specifier is 'X', the extended digits are all digits together with the letters A through F.

A bit string literal has a value that is a string literal consisting of the character literals '0' and '1'. If the base specifier is 'B', the value of the bit string literal is the sequence given explicitly by the bit value itself after any underlines have been removed.

If the base specifier is 'O' (respectively 'X'), the value of the bit string literal is the sequence obtained by replacing each extended digit in the bit_value by a sequence consisting of the three (respectively four) values representing that extended digit taken from the character literals '0' and '1';as in the case of the base specifier 'B', underlines are first removed. Each extended digit is replaced according to the table on the following page:

Extended digit	Replacement when the base specifier is 'O'	Replacement when the base specifier is 'X'
0	000	0000
1	001	0001
2	010	0010
3	011	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111
8	(illegal)	1000
9	(illegal)	1001
A	(illegal)	1010
B	(illegal)	1011
C	(illegal)	1100
D	(illegal)	1101
E	(illegal)	1110

```
      F                   (illegal)                          1111
```

The *length* of a bit string literal is the length of its string literal value.

Example:

```
     B"1111_1111_1111"          --  Equivalent to the string literal "111111111111"
     X"FFF"                     --  Equivalent to B"1111_1111_1111"
     O"777"                     --  Equivalent to B"111_111_111"
     X"777"                     --  Equivalent to B"0111_0111_0111"

     constant c1: STRING := B"1111_1111_1111";

     constant c2: BIT_VECTOR := X"FFF";

     type MVL is ('X', '0', '1', 'Z');
     type MVL_VECTOR is array (NATURAL range <>) of MVL;
     constant c3: MVL_VECTOR := O"777";

     assert    c1'LENGTH = 12 and
               c2'LENGTH = 12 and
               c3 = "111111111";
```

13.8 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a VHDL description. The presence or absence of comments has no influence on whether a description is legal or illegal. Furthermore, comments do not influence the execution of a simulation module; their sole purpose is to enlighten the human reader.

Examples:

```
     --  The last sentence above echoes the Algol 68 report.

     end;                                    -- Processing of LINE is complete

     --  A long comment may be split onto
     --  two or more consecutive lines.

     -----------  The first two hyphens start the comment.
```

NOTE--Horizontal tabulation can be used in comments, after the double hyphen, and is equivalent to one or more spaces (SPACE characters) (see 13.2).

13.9 Reserved words

The identifiers listed below are called *reserved words* and are reserved for significance in the language. For readability of this manual, the reserved words appear in lowercase boldface.

abs	**file**	**nand**	**select**

access	**for**	**new**	**severity**
after	**function**	**next**	**signal**
alias		**nor**	**shared**
all	**generate**	**not**	**sla**
and	**generic**	**null**	**sll**
architecture	**group**		**sra**
array	**guarded**	**of**	**srl**
assert		**on**	**subtype**
attribute	**if**	**open**	
	impure	**or**	**then**
begin	**in**	**others**	**to**
block	**inertial**	**out**	**transport**
body	**inout**		**type**
buffer	**is**	**package**	
bus		**port**	**unaffected**
	label	**postponed**	**units**
case	**library**	**procedure**	**until**
component	**linkage**	**process**	**use**
configuration	**literal**	**pure**	
constant	**loop**		**variable**
		range	
disconnect	**map**	**record**	**wait**
downto	**mod**	**register**	**when**
		reject	**while**
else		**rem**	**with**
elsif		**report**	
end		**return**	**xnor**
entity		**rol**	**xor**
exit		**ror**	

A reserved word must not be used as an explicitly declared identifier.

NOTES

1--Reserved words differing only in the use of corresponding upper and lowercase letters are considered as the same (see 13.3.1). The reserved word **range** is also used as the name of a predefined attribute.

2--An extended identifier whose sequence of characters inside the leading and trailing backslashes is identical to a reserved word is not a reserved word. For example, \next\ is a legal (extended) identifier and is not the reserved word **next**.

13.10 Allowable replacements of characters

The following replacements are allowed for the vertical line, number sign, and quotation mark basic characters:

```
    -- A vertical line (|) can be replaced by an exclamation mark (!) where used as
    a delimiter.

    -- The number sign (#) of a based literal can be replaced by colons
    (:),provided that the replacement is done for both occurrences.

    -- The quotation marks (") used as string brackets at both ends of a string
    literal can be replaced by percent signs (%), provided that the enclosed
    sequence of characters contains no quotation marks, and provided that both
    string brackets are replaced. Any percent sign within the sequence of
    characters must then be doubled, and each such doubled percent sign is
    interpreted as a single percent sign value. The same replacement is allowed for
    a bit string literal, provided that both bit string brackets are replaced.
```

These replacements do not change the meaning of the description.

NOTES

1--It is recommended that use of the replacements for the vertical line,number sign, and quotation marks be restricted to cases where the corresponding graphical symbols are not available. Note that the vertical line appears as a broken line on some equipment; replacement is not recommended in this case.

2--The rules given for identifiers and abstract literals are such that lowercase and uppercase letters can be used indifferently; these lexical elements can thus be written using only characters of the basic character set.

Annex A

Syntax summary

(informative)

This annex provides a summary of the syntax for VHDL. Productions are ordered alphabetically by left-hand nonterminal name. The clause number indicates the clause where the production is given.

```
     abstract_literal ::=  decimal_literal | based_literal
[§ 13.4]

     access_type_definition ::=  access subtype_indication
[§ 3.3  ]

     actual_designator ::=
[§ 4.3.2.2]
           expression
         | signal_name
         | variable_name
         | file_name
         | open

     actual_parameter_part ::=  parameter_association_list
[§ 7.3.3]

     actual_part ::=
[§ 4.3.2.2]
           actual_designator
         | function_name ( actual_designator )
         | type_mark ( actual_designator )

     adding_operator ::=  + | -  | &
[§ 7.2]

     aggregate ::=
[§ 7.3.2]
          ( element_association { , element_association } )

     alias_declaration ::=
[§ 4.3.3]
           alias alias_designator [ : subtype_indication ] is name [ signature ] ;

     alias_designator ::=  identifier | character_literal | operator_symbol
[§  4.3.3]

     allocator ::=
[§ 7.3.6]
```

```
            new subtype_indication
          | new qualified_expression

    architecture_body ::=
[§ 1.2]
            architecture identifier of entity_name is
                architecture_declarative_part
            begin
                architecture_statement_part
            end [ architecture ] [ architecture_simple_name ] ;

    architecture_declarative_part ::=
[§ 1.2.1]
          { block_declarative_item }

    architecture_statement_part ::=
[§ 1.2.2]
          { concurrent_statement }

    array_type_definition ::=
[§ 3.2.1]
          unconstrained_array_definition  |  constrained_array_definition

    assertion ::=
[§ 8.2]
         assert condition
             [ report expression ]
             [ severity expression ]

    assertion_statement ::=  [ label : ] assertion ;
[§ 8.2]

    association_element ::=
[§ 4.3.2.2]
         [ formal_part => ] actual_part

    association_list ::=
[§ 4.3.2.2]
         association_element { , association_element }

    attribute_declaration :
[§ 4.4]
          attribute identifier : type_mark ;

    attribute_designator ::=  attribute_simple_name
[§ 6.6]

    attribute_name ::=
[§ 6.6]
          prefix [ signature ] ' attribute_designator [ ( expression ) ]

    attribute_specification ::=
[§ 5.1]
          attribute attribute_designator of entity_specification is expression ;

    base ::=  integer
```

```
[§ 13.4.2]

     base_specifier ::=  B | O | X
[§ 13.7]

     based_integer ::=
[§ 13.4.2]
          extended_digit { [ underline ] extended_digit }

     based_literal ::=
[§ 13.4.2]
          base # based_integer [ . based_integer ] # [ exponent ]

     basic_character ::=
[§ 13.1]
          basic_graphic_character | format_effector

     basic_graphic_character ::=
[§ 13.1]
          upper_case_letter | digit | special_character| space_character

     basic_identifier ::=  letter  { [ underline ] letter_or_digit }
[§ 13.3.1]

     binding_indication ::=
[§ 5.2.1]
          [ use entity_aspect ]
          [ generic_map_aspect ]
          [ port_map_aspect ]

     bit_string_literal ::=  base_specifier " [ bit_value ] "
[§ 13.7]

     bit_value ::=  extended_digit { [ underline ] extended_digit }
[§ 13.7]

     block_configuration ::=
[§ 1.3.1]
           for block_specification
                 { use_clause }
                 { configuration_item }
           end for ;

     block_declarative_item ::=
[§ 1.2.1]
            subprogram_declaration
         | subprogram_body
         | type_declaration
         | subtype_declaration
         | constant_declaration
         | signal_declaration
         | shared_variable_declaration
         | file_declaration
         | alias_declaration
         | component_declaration
         | attribute_declaration
```

```
        | attribute_specification
        | configuration_specification
        | disconnection_specification
        | use_clause
        | group_template_declaration
        | group_declaration

    block_declarative_part ::=
[§ 9.1]

        { block_declarative_item }

    block_header ::=
[§ 9.1]
        [ generic_clause
        [ generic_map_aspect ; ] ]
        [ port_clause
        [ port_map_aspect ; ] ]

    block_specification ::=
[§ 1.3.1]
          architecture_name
        | block_statement_label
        | generate_statement_label [ ( index_specification ) ]

    block_statement ::=
[§ 9.1]
          block_label :
              block [ ( guard_expression ) ] [ is ]
                  block_header
                  block_declarative_part
              begin
                  block_statement_part
              end block [ block_label ] ;

    block_statement_part ::=
[§ 9.1]
        { concurrent_statement }

    case_statement ::=
[§ 8.8]
        [ case_label : ]
            case expression is
                 case_statement_alternative
                 { case_statement_alternative }
            end case [ case_label ] ;

    case_statement_alternative ::=
[§ 8.8]
         when choices =>
            sequence_of_statements

    character_literal ::= ' graphic_character '
[§ 13.5]

    choice ::=
```

```
[§ 7.3.2]
           simple_expression
         | discrete_range
         | element_simple_name
         | others

     choices ::=  choice { | choice }
[§ 7.3.2]

     component_configuration ::=
[§ 1.3.2]
         for component_specification
              [ binding_indication ; ]
              [ block_configuration ]
          end for ;

     component_declaration ::=
[§ 4.5]
         component identifier [ is ]
            [ local_generic_clause ]
            [ local_port_clause ]
         end component [ component_simple_name ] ;

     component_instantiation_statement ::=
[§ 9.6]
         instantiation_label :
             instantiated_unit
                  [ generic_map_aspect ]
                  [ port_map_aspect ] ;

     component_specification ::=
[§ 5.2]

         instantiation_list : component_name

     composite_type_definition ::=
[§ 3.2]
           array_type_definition
         | record_type_definition

     concurrent_assertion_statement ::=
[§ 9.4]
         [ label : ] [ postponed ] assertion ;

     concurrent_procedure_call_statement ::=
[§ 9.3]
         [ label : ] [ postponed ] procedure_call ;

     concurrent_signal_assignment_statement ::=
[§ 9.5]
           [ label : ] [ postponed ] conditional_signal_assignment
         | [ label : ] [ postponed ] selected_signal_assignment

     concurrent_statement ::=
[§ 9]
           block_statement
```

```
        | process_statement
        | concurrent_procedure_call_statement
        | concurrent_assertion_statement
        | concurrent_signal_assignment_statement
        | component_instantiation_statement
        | generate_statement

    condition ::=  boolean_expression
[§ 8.1]

    condition_clause ::=  until condition
[§ 8.1]

    conditional_signal_assignment ::=
[§ 9.5.1]
         target  <=  options conditional_waveforms ;

    conditional_waveforms ::=
[§ 9.5.1]

         { waveform when condition else }
           waveform [ when condition ]

    configuration_declaration ::=
[§ 1.3]
          configuration identifier of entity_name is
               configuration_declarative_part
               block_configuration
          end [ configuration ] [ configuration_simple_name ] ;

    configuration_declarative_item ::=
[§ 1.3]
           use_clause
        | attribute_specification
        | group_declaration

    configuration_declarative_part ::=
[§ 1.3]
       { configuration_declarative_item }

    configuration_item ::=
[§ 1.3.1]
          block_configuration
         | component_configuration

    configuration_specification ::=
[§ 5.2]
         for component_specification binding_indication ;

    constant_declaration ::=
[§ 4.3.1.1]
         constant identifier_list : subtype_indication [ := expression ] ;

    constrained_array_definition ::=
[§ 3.2.1]
         array index_constraint of element_subtype_indication
```

```
     constraint ::=
[§ 4.2]
           range_constraint
         | index_constraint

     context_clause ::=  { context_item }
[§ 11.3]

     context_item ::=
[§ 11.3]
          library_clause
        | use_clause

     decimal_literal ::=  integer [ . integer ] [ exponent ]
[§ 13.4.1]

     declaration ::=
[§ 4]
           type_declaration
         | subtype_declaration
         | object_declaration
         | interface_declaration
         | alias_declaration
         | attribute_declaration
         | component_declaration
         | group_template_declaration
         | group_declaration
         | entity_declaration
         | configuration_declaration
         | subprogram_declaration
         | package_declaration

     delay_mechanism ::=
[§ 8.4]
            transport
        | [ reject time_expression ] inertial

     design_file ::=  design_unit { design_unit }
[§ 11.1]

     design_unit ::=  context_clause library_unit
[§ 11.1]

     designator ::=  identifier  |  operator_symbol
[§ 2.1]

     direction ::=  to | downto
[§ 3.1]

     disconnection_specification ::=
[§ 5.3]

        disconnect guarded_signal_specification after time_expression ;

     discrete_range ::=  discrete_subtype_indication | range
```

```
[§ 3.2.1]

     element_association ::=
[§ 7.3.2]
        [ choices => ] expression

     element_declaration ::=
[§ 3.2.2]
         identifier_list : element_subtype_definition ;

     element_subtype_definition ::=  subtype_indication
[§ 3.2.2]

     entity_aspect ::=
[§ 5.2.1.1]
           entity entity_name [ ( architecture_identifier) ]
         | configuration configuration_name
         | open

     entity_class ::=
[§ 5.1]
           entity      | architecture   | configuration
         | procedure   | function       | package
         | type        | subtype        | constant
         | signal      | variable       | component
         | label       | literal        | units
         | group       | file

     entity_class_entry ::=  entity_class [ <> ]
[§ 4.6]

     entity_class_entry_list ::=
[§ 4.6]
          entity_class_entry { , entity_class_entry }

     entity_declaration ::=
[§ 1.1]
          entity identifier is
              entity_header
              entity_declarative_part
      [   begin
              entity_statement_part ]
          end [ entity ] [ entity_simple_name ] ;

     entity_declarative_item ::=
[§ 1.1.2]
           subprogram_declaration
         | subprogram_body
         | type_declaration
         | subtype_declaration
         | constant_declaration
         | signal_declaration
         | shared_variable_declaration
         | file_declaration
         | alias_declaration
         | attribute_declaration
```

```
        | attribute_specification
        | disconnection_specification
        | use_clause
        | group_template_declaration
        | group_declaration

    entity_declarative_part ::=
[§ 1.1.2]
       { entity_declarative_item }

    entity_designator ::=  entity_tag [ signature ]
[§ 5.1]

    entity_header ::=
[§ 1.1.1]
        [ formal_generic_clause ]
        [ formal_port_clause ]

    entity_name_list ::=
[§ 5.1]
          entity_designator { , entity_designator }
        | others
        | all

    entity_specification ::=
[§ 5.1]
        entity_name_list : entity_class

    entity_statement ::=
[§ 1.1.3]

          concurrent_assertion_statement
        | passive_concurrent_procedure_call_statement
        | passive_process_statement

    entity_statement_part ::=
[§ 1.1.3]
        { entity_statement }

    entity_tag ::=  simple_name | character_literal | operator_symbol
[§   5.1]

    enumeration_literal ::=  identifier | character_literal
[§ 3.1.1]

    enumeration_type_definition ::=
[§ 3.1.1]
        ( enumeration_literal { , enumeration_literal } )

    exit_statement ::=
[§ 8.11]
        [ label : ] exit [ loop_label ] [ when condition ] ;

    exponent ::=  E [ + ] integer | E - integer
[§ 13.4.1]

    expression ::=
```

```
[§ 7.1]
            relation { and relation }
          | relation { or relation }
          | relation { xor relation }
          | relation [ nand relation ]
          | relation [ nor relation ]
          | relation { xnor relation }

     extended_digit ::=  digit | letter
[§ 13.4.2]

     extended_identifier ::=  \ graphic_character { graphic_character } \
[§ 13.3.2]

     factor ::=
[§ 7.1]
            primary [ ** primary ]
          | abs primary
          | not primary

     file_declaration ::=
[§ 4.3.1.4]
          file identifier_list : subtype_indication [ file_open_information ] ;

     file_logical_name ::=  string_expression
[§ 4.3.1.4]

     file_open_information ::=
[§ 4.3.1.4]
           [ open file_open_kind_expression ] is file_logical_name

     file_type_definition ::=
[§ 3.4]
           file  of type_mark

     floating_type_definition  ::=  range_constraint
[§ 3.1.4]

     formal_designator ::=
[§ 4.3.2.2]
            generic_name
          | port_name
          | parameter_name

     formal_parameter_list ::=  parameter_interface_list
[§ 2.1.1]

     formal_part ::=
[§ 4.3.2.2]
            formal_designator
          | function_name ( formal_designator )
          | type_mark ( formal_designator )

     full_type_declaration ::=
[§ 4.1]
         type identifier is type_definition ;
```

```
     function_call ::=
[§ 7.3.3]

        function_name [ ( actual_parameter_part ) ]

     generate_statement ::=
[§ 9.7]
        generate_label :
            generation_scheme generate
                [ { block_declarative_item }
            begin ]
                { concurrent_statement }
            end generate [ generate_label ] ;

     generation_scheme ::=
[§ 9.7]
            for generate_parameter_specification
          | if condition

     generic_clause ::=
[§ 1.1.1]
         generic ( generic_list ) ;

     generic_list ::=  generic_interface_list
[§ 1.1.1.1]

     generic_map_aspect ::=
[§ 5.2.1.2]
         generic map ( generic_association_list )

     graphic_character ::=
[§ 13.1]
         basic_graphic_character | lower_case_letter | other_special_character

     group_constituent ::=  name | character_literal
[§ 4.7]

     group_constituent_list ::=  group_constituent { , group_constituent}
[§ 4.7]

     group_declaration ::=
[§ 4.7]
         group identifier : group_template_name ( group_constituent_list ) ;

     group_template_declaration ::=
[§ 4.6]
         group identifier is ( entity_class_entry_list ) ;

     guarded_signal_specification ::=
[§ 5.3]
         guarded_signal_list : type_mark

     identifier ::=  basic_identifier | extended_identifier
[§ 13.3]

     identifier_list ::=  identifier { , identifier }
```

```
[§ 3.2.2]

     if_statement ::=
[§ 8.7]
          [ if_label : ]
               if condition then
                    sequence_of_statements
               { elsif condition then
                    sequence_of_statements }
               [ else
                    sequence_of_statements ]
               end if [ if_label ] ;

     incomplete_type_declaration ::=  type identifier ;
[§ 3.3.1]

     index_constraint ::=  ( discrete_range { , discrete_range } )
[§ 3.2.1]

     index_specification ::=
[§;   1.3.1]
          discrete_range
        | static_expression

     index_subtype_definition ::= type_mark range <>
[§ 3.2.1]

     indexed_name ::=  prefix ( expression { , expression } )
[§ 6.4]

     instantiated_unit ::=
[§ 9.6]

          [ component ] component_name
        | entity entity_name [ ( architecture_identifier ) ]
        | configuration configuration_name

     instantiation_list ::=
[§ 5.2]
          instantiation_label { , instantiation_label }
        | others
        | all

     integer ::=  digit  { [ underline ] digit }
[§ 13.4.1]

     integer_type_definition ::=  range_constraint
[§ 3.1.2]

     interface_constant_declaration ::=
[§ 4.3.2]
          [ constant ] identifier_list : [ in ] subtype_indication [ :=
static_expression ]

     interface_declaration ::=
[§ 4.3.2]
```

```
            interface_constant_declaration
          | interface_signal_declaration
          | interface_variable_declaration
          | interface_file_declaration

     interface_element ::=  interface_declaration
[§ 4.3.2.1]

     interface_file_declaration ::=
[§ 4.3.2]
          file identifier_list : subtype_indication

     interface_list ::=
[§ 4.3.2.1]
          interface_element { ; interface_element }

     interface_signal_declaration ::
[§ 4.3.2]
          [signal] identifier_list : [ mode ] subtype_indication [ bus ] [ :=
static_expression ]

     interface_variable_declaration ::=
[§ 4.3.2]
          [variable] identifier_list : [ mode ] subtype_indication [ :=
static_expression ]

     iteration_scheme ::=
[§ 8.9]
             while condition
          | for loop_parameter_specification

     label ::=  identifier
[§ 9.7]

     letter ::=  upper_case_letter | lower_case_letter
[§;   13.3.1]

     letter_or_digit ::=  letter | digit
[§ 13.3.1]

     library_clause ::=  library logical_name_list ;
[§ 11.2]

     library_unit ::=
[§ 11.1]

             primary_unit
          | secondary_unit

     literal ::=
[§ 7.3.1]
             numeric_literal
          | enumeration_literal
          | string_literal
          | bit_string_literal
          | null
```

```
     logical_name ::=  identifier
[§ 11.2]

     logical_name_list ::=  logical_name { , logical_name }
[§   11.2]

     logical_operator ::=  and | or | nand | nor | xor | xnor
[§; 7.2]

     loop_statement ::=
[§ 8.9]
         [ loop_label : ]
              [ iteration_scheme ] loop
                   sequence_of_statements
              end loop [ loop_label ] ;

     miscellaneous_operator ::=  ** | abs | not
[§ 7.2]

     mode ::=  in | out | inout | buffer | linkage
[§ 4.3.2]

     multiplying_operator ::=  * | / | mod | rem
[§ 7.2]

     name ::=
[§ 6.1]
            simple_name
         | operator_symbol
         | selected_name
         | indexed_name
         | slice_name
         | attribute_name
     next_statement ::=
[§ 8.10]
         [ label : ] next [ loop_label ] [ when condition ] ;

     null_statement ::=  [ label : ] null ;
[§ 8.13]

     numeric_literal ::=
[§ 7.3.1]
            abstract_literal
         | physical_literal

     object_declaration ::=
[§ 4.3.1]
            constant_declaration
         | signal_declaration
         | variable_declaration
         | file_declaration

     operator_symbol ::=  string_literal
[§ 2.1]

     options ::=  [ guarded ] [ delay_mechanism ]
```

```
[§ 9.5]

     package_body ::=
[§ 2.6]
          package body package_simple_name is
              package_body_declarative_part
          end [ package body ] [ package_simple_name ] ;

     package_body_declarative_item ::=
[§ 2.6]
             subprogram_declaration
          | subprogram_body
          | type_declaration
          | subtype_declaration
          | constant_declaration
          | shared_variable_declaration
          | file_declaration
          | alias_declaration
          | use_clause
          | group_template_declaration
          | group_declaration

     package_body_declarative_part ::=
[§ 2.6]

          { package_body_declarative_item }

     package_declaration ::=
[§ 2.5]
          package identifier is
             package_declarative_part
          end [ package ] [ package_simple_name ] ;

     package_declarative_item ::=
[§ 2.5]
             subprogram_declaration
          | type_declaration
          | subtype_declaration
          | constant_declaration
          | signal_declaration
          | shared_variable_declaration
          | file_declaration
          | alias_declaration
          | component_declaration
          | attribute_declaration
          | attribute_specification
          | disconnection_specification
          | use_clause
          | group_template_declaration
          | group_declaration

     package_declarative_part ::=
[§ 2.5]
          { package_declarative_item }

     parameter_specification ::=
```

```
[§ 8.9]
        identifier in discrete_range

    physical_literal ::=  [ abstract_literal ] unit_name
[§ 3.1.3]

    physical_type_definition ::=
[§ 3.1.3]
        range_constraint
            units
                primary_unit_declaration
                { secondary_unit_declaration }
            end units [ physical_type_simple_name ]

    port_clause ::=
[§ 1.1.1]
        port ( port_list ) ;

    port_list ::=  port_interface_list
[§ 1.1.1.2]

    port_map_aspect ::=
[§ 5.2.1.2]
        port map ( port_association_list )

    prefix ::=
[§ 6.1]
           name
        | function_call

    primary ::=
[§ 7.1]
           name
        | literal
        | aggregate
        | function_call
        | qualified_expression
        | type_conversion
        | allocator
        | ( expression )

    primary_unit ::
[§ 11.1]
           entity_declaration
        | configuration_declaration
        | package_declaration

    primary_unit_declaration ::= identifier;
    procedure_call ::=  procedure_name [ ( actual_parameter_part ) ]
[§ 8.6]

    procedure_call_statement ::=  [ label : ]  procedure_call ;
[§ 8.6]

    process_declarative_item ::=
[§ 9.2]
```

```
            subprogram_declaration
          | subprogram_body
          | type_declaration
          | subtype_declaration
          | constant_declaration
          | variable_declaration
          | file_declaration
          | alias_declaration
          | attribute_declaration
          | attribute_specification
          | use_clause
          | group_template_declaration
          | group_declaration

     process_declarative_part ::=
[§;   9.2]
          { process_declarative_item }

     process_statement ::=
[§ 9.2]
          [ process_label : ]
               [ postponed ] process [ ( sensitivity_list ) ] [ is ]
                    process_declarative_part
               begin
                    process_statement_part
               end [ postponed ] process [ process_label ] ;

     process_statement_part ::=
[§ 9.2]
          { sequential_statement }

     qualified_expression ::=
[§ 7.3.4]
             type_mark ' ( expression )
          | type_mark ' aggregate
     range ::=
[§ 3.1]
             range_attribute_name
          | simple_expression direction simple_expression

     range_constraint ::=  range range
[§ 3.1]

     record_type_definition ::=
[§ 3.2.2]
          record
               element_declaration
              { element_declaration }
          end record [ record_type_simple_name ]

     relation ::=
[§ 7.1]
          shift_expression [ relational_operator shift_expression ]

     relational_operator ::=   =  |  /=  |  <  |  <=  |  >  |  >=
[§ 7.2]
```

```
     report_statement ::=
[§ 8.3]
         [ label : ]
              report expression
               [ severity expression ] ;

     return_statement ::=
[§ 8.12]
        [ label : ] return [ expression ] ;

     scalar_type_definition ::=
[§ 3.1]
           enumeration_type_definition  | integer_type_definition
         | floating_type_definition          | physical_type_definition

     secondary_unit ::=
[§ 11.1]
           architecture_body
        | package_body

     secondary_unit_declaration ::=  identifier = physical_literal ;
[§ 3.1.3]

     selected_name ::=  prefix . suffix
[§ 6.3]

     selected_signal_assignment ::=
[§ 9.5.2]

         with expression select
              target <= options selected_waveforms ;

     selected_waveforms ::=
[§ 9.5.2]
         { waveform when choices , }
           waveform when choices

     sensitivity_clause ::=  on sensitivity_list
[§ 8.1]

     sensitivity_list ::=  signal_name { , signal_name }
[§ 8.1]

     sequence_of_statements ::=
[§ 8]
         { sequential_statement }

     sequential_statement ::=
[§ 8]
           wait_statement
        | assertion_statement
        | report_statement
        | signal_assignment_statement
        | variable_assignment_statement
        | procedure_call_statement
        | if_statement
```

```
        | case_statement
        | loop_statement
        | next_statement
        | exit_statement
        | return_statement
        | null_statement

    shift_expression ::=
[§ 7.1]
         simple_expression [ shift_operator simple_expression ]

    shift_operator ::=  sll | srl | sla | sra | rol | ror
[§ 7.2]

    sign ::=  + | -
[§ 7.2]

    signal_assignment_statement ::=
[§ 8.4]
        [ label : ] target <= [ delay_mechanism ] waveform ;

    signal_declaration ::=
[§ 4.3.1.2]
        signal identifier_list : subtype_indication [ signal_kind ] [ := expression
] ;

    signal_kind ::=  register  |  bus
[§;   4.3.1.2]

    signal_list ::=
[§ 5.3]
           signal_name { , signal_name }
        | others
        | all

    signature ::=  [ [ type_mark { , type_mark } ] [ return type_mark ] ]
[§ 2.3.2]

    simple_expression ::=
[§ 7.1]
        [ sign ] term { adding_operator term }

    simple_name ::=  identifier
[§ 6.2]

    slice_name ::=  prefix ( discrete_range )
[§ 6.5]

    string_literal ::=  " { graphic_character } "
[§ 13.6]

    subprogram_body ::=
[§ 2.2]
        subprogram_specification is
            subprogram_declarative_part
        begin
            subprogram_statement_part
```

```
         end [ subprogram_kind ] [ designator ] ;

     subprogram_declaration ::=
[§ 2.1]

        subprogram_specification ;

     subprogram_declarative_item ::=
[§ 2.2]
            subprogram_declaration
          | subprogram_body
          | type_declaration
          | subtype_declaration
          | constant_declaration
          | variable_declaration
          | file_declaration
          | alias_declaration
          | attribute_declaration
          | attribute_specification
          | use_clause
          | group_template_declaration
          | group_declaration

     subprogram_declarative_part ::=
[§ 2.2]
        { subprogram_declarative_item }

     subprogram_kind ::=  procedure | function
[§ 2.2]

     subprogram_specification ::=
[§ 2.1]
           procedure designator [ ( formal_parameter_list ) ]
        | [ pure | impure ] function designator [ ( formal_parameter_list ) ]
                  return type_mark

     subprogram_statement_part ::=
[§ 2.2]
        { sequential_statement }

     subtype_declaration ::=
[§ 4.2]
        subtype identifier is subtype_indication ;

     subtype_indication ::=
[§ 4.2]
        [ resolution_function_name ] type_mark [ constraint ]

     suffix ::=
[§ 6.3]
            simple_name
          | character_literal
          | operator_symbol
          | all

     target ::=
```

```
[§ 8.4]
          name
          | aggregate

     term ::=
[§ 7.1]
          factor { multiplying_operator factor }

     timeout_clause ::=  for time_expression
[§ 8.1]

     type_conversion ::=  type_mark ( expression )
[§ 7.3.5]

     type_declaration ::=
[§ 4.1]
             full_type_declaration
          | incomplete_type_declaration

     type_definition ::=
[§ 4.1]
            scalar_type_definition
          | composite_type_definition
          | access_type_definition
          | file_type_definition

     type_mark ::=
[§ 4.2]
             type_name
          | subtype_name

     unconstrained_array_definition ::=
[§ 3.2.1]

          array ( index_subtype_definition { , index_subtype_definition } )
               of element_subtype_indication

     use_clause ::=
[§ 10.4]
          use selected_name { , selected_name } ;

     variable_assignment_statement ::=
[§ 8.5]
          [ label : ] target  := expression ;

     variable_declaration ::=
[§ 4.3.1.3]
          [ shared ] variable identifier_list : subtype_indication [ := expression ] ;

     wait_statement ::=
[§ 8.1]
          [ label : ] wait [ sensitivity_clause ] [ condition_clause ] [
timeout_clause ] ;

     waveform ::=
[§ 8.4]
```

```
          waveform_element { , waveform_element }
        | unaffected

    waveform_element ::=
[§ 8.4.1]
          value_expression [ after time_expression ]
        | null [ after time_expression ]
```

Annex B

Glossary

(informative)

This glossary contains brief, informal descriptions for a number of terms and phrases used to define this language. The complete, formal definition of each term or phrase is provided in the main body of the standard.

For each entry, the relevant clause numbers in the text are given. Some descriptions refer to multiple clauses in which the single concept is discussed; for these, the clause number containing the definition of the concept is given in italics. Other descriptions contain multiple clause numbers when they refer to multiple concepts; for these, none of the clause numbers are italicized.

B.1 abstract literal: A literal of the *universal_real* abstract type or the *universal_integer* abstract type. (§ 13.2 , *§* 13.4)

B.2 access type: A type that provides access to an object of a given type. Access to such an object is achieved by an access value returned by an allocator; the access value is said to *designate* the object.(*§*;3, *§* 3.3)

B.3 access mode: The mode in which a file object is opened, which can be either *read-only* or *write-only*. The access mode depends on the value supplied to the Open_Kind parameter. (*§* 3.4.1 , § 14.3).

B.4 access value: A value of an access type. This value is returned by an allocator and designates an object (which must be a variable) of a given type. A null access value designates no object. An access value can only designate an object created by an allocator; it cannot designate an object declared by an object declaration. (*§3*, § 3.3)

B.5 active driver: A driver that acquires a new value during a simulation cycle regardless of

whether the new value is different from the previous value. (§ 12.6.2 , § 12.6.4)

B.6 actual: An expression, a port, a signal, or a variable associated with a formal port, formal parameter, or formal generic. (§ 1.1.1.1 , *§ 1.1.1.2* , § 3.2.1.1 , § 4.3.1.2 , *§ 4.3.2.2* , § 5.2.1 , *§ 5.2.1.2*)

B.7 aggregate:

a) The kind of expression, denoting a value of a composite type. The value is specified by giving the value of each of the elements of the composite type. Either a positional association or a named association may be used to indicate which value is associated with which element.

b) A kind of target of a variable assignment statement or signal assignment statement assigning a composite value. The target is then said to *be in the form of an aggregate*. (§ 7.3.1 , *§ 7.3.2* , § 7.3.4 , § 7.3.5 , § 7.5.2)

B.8 alias: An alternate name for a named entity. (§ 4.3.3)

B.9 allocator: An operation used to create anonymous, variable objects accessible by means of *access values*. (*§* 3.3 , § 7.3.6)

B.10 analysis: The syntactic and semantic analysis of source code in a VHDL design file and the insertion of intermediate form representations of design units into a design library. (§ 11.1 , § 11.2 , § 11.4)

B.11 anonymous: The undefined simple name of an item, which is created implicitly. The base type of a numeric type or an array type is anonymous;similarly, the object denoted by an access value is anonymous. (§ 4.1)

B.12 appropriate: A prefix is said to be appropriate for a type if the type of the prefix is the type considered, or if the type of the prefix is an access type whose designated type is the type considered.(§ 6.1)

B.13 architecture body: A body associated with an entity declaration to describe the internal organization or operation of a design entity. An architecture body is used to describe the behavior, data flow, or structure of a design entity. (§1, *§* 1.2)

B.14 array object: An object of an array type.(*§*3*)*

B.15 array type: A type, the value of which consists of elements that are all of the same subtype (and hence, of the same type). Each element is uniquely distinguished by an index (for a one-dimensional array) or by a sequence of indexes (for a multidimensional array). Each index must be a value of a discrete type and must lie in the correct index range. (§ 3.2.1)

B.16 ascending range: A range L **to** R. (§ 3.1)

B.17 ASCII: The American Standard Code for Information Interchange. The package Standard contains the definition of the type Character, the first 128 values of which represent the ASCII character set. (§ 3.1.1 , *§*; 14.2)

B.18 assertion violation: A violation that occurs when the condition of an assertion statement evaluates to false. (§ 8.2)

B.19 associated driver: The single driver for a signal in the (explicit or equivalent) process statement containing the signal assignment statement.(12.6.1 ">§ 12.6.1)

B.20 associated in whole: When a single association element of a composite formal supplies the association for the entire formal.(§ 4.3.2.2)

B.21 associated individually: A property of a formal port, generic, or parameter of a composite type with respect to some association list. A composite formal whose association is defined by multiple association elements in a single association list is said to be *associated individually* in that list. The formats of such association elements must denote non-overlapping subelements or slices of the formal. (§ 4.3.2.2)

B.22 association element: An element that associates an actual or local with a local or formal. (§ 4.3.2.2)

B.23 association list: A list that establishes correspondences between formal or local port or parameter names and local or actual names or expressions. (§ 4.3.2.2)

B.24 attribute: A definition of some characteristic of a named entity. Some attributes are predefined for types, ranges, values, signals, and functions. The remaining attributes are user defined and are always constants.(§ 4.4)

B.25 base specifier: A lexical element that indicates whether a bit string literal is to be interpreted as a binary, octal, or hexadecimal value.(§ 13.7)

B.26 base type: The type from which a subtype defines a subset of possible values, otherwise

known as a *constraint*. This subset is not required to be proper. The base type of a type is the type itself. The base type of a subtype is found by recursively examining the type mark in the subtype indication defining the subtype. If the type mark denotes a type, that type is the base type of the subtype; otherwise, the type mark is a subtype,and this procedure is repeated on that subtype. (§3) *See also* **subtype**.

B.27 based literal: An abstract literal expressed in a form that specifies the base explicitly. The base is restricted to the range 2 to 16.(§ 13.4.2)

B.28 basic operation: An operation that is inherent in one of the following:

a) An assignment (in an assignment statement or initialization);

b) An allocator;

c) A selected name, an indexed name, or a slice name;

d) A qualification (in a qualified expression), an explicit type conversion, a formal or actual designator in the form of a type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type; or

e) A numeric literal (for a universal type), the literal null (for an access type), a string literal, a bit string literal, an aggregate, or a predefined attribute. (§3)

B.29 basic signal: A signal that determines the driving values for all other signals. A basic signal is

-- Either a scalar signal or a resolved signal;

-- Not a subelement of a resolved signal;

-- Not an implicit signal of the form S'Stable(T), S'Quiet(T), or S'Transaction; and

-- Not an implicit signal GUARD. (§ 12.6.2)

B.30 belong (to a range): A property of a value with respect to some range. The value V is said to *belong to a range* if the relations (lower bound <= V) and (V <= upper bound) are both true,where lower bound and upper bound are the lower and upper bounds, respectively,of the range. (*§ 3.1* , § 3.2.1)

B. 31 belong (to a subtype): A property of a value with respect to some subtype. A value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the applicable constraint.(*§3*, § 3.2.1)

B.32 binding: The process of associating a design entity and,optionally, an architecture with an instance of a component. A binding can be specified in an explicit or a default binding indication. (§ 1.3 , *§* 5.2.1 , § 5.2.2 , § **12.3.2.2** , § 12.4.3)

B.33 bit string literal: A literal formed by a sequence of extended digits enclosed between two quotation (") characters and preceded by a base specifier. The type of a bit string literal is determined from the context.(§ 7.3.1 , *§* 13.7 *)*

B. 34 block: The representation of a portion of the hierarchy of a design. A block is either an external block or an internal block.(*§1*, § 1.1.1.1 , § 1.1.1.2 , § 1.2.1 § 1.3 , *§ 1.3.1* , § 1.3.2)

B.35 bound: A label that is identified in the instantiation list of a configuration specification. (§ 5.2)

B.36 box: The symbol <> in an index subtype definition, which stands for an undefined range. Different objects of the type need not have the same bounds and direction. (§ 3.2.1)

B.37 bus: One kind of guarded signal. A bus floats to a user-specified value when all of its drivers are turned off. (§ 4.3.1.2 , § 4.3.2)

B.38 character literal: A literal of the character type. Character literals are formed by enclosing one of the graphic characters (including the space and nonbreaking space characters) between two apostrophe (') characters.(§ 13.2 , *§ 13.5)*

B.39 character type: An enumeration type with at least one character literal among its enumeration literals. (*§* 3.1.1 , § 3.1.1.1)

B.40 closely related types: Two type marks that denote the same type or two numeric types. Two array types may also be closely related if they have the same dimensionality, if their index types at each position are closely related, and if the array types have the same element types. Explicit type conversion is only allowed between closely related types. (§ 7.3.5)

B.41 complete: A loop that has finished executing. Similarly, an iteration scheme of a loop is complete when the condition of a while iteration scheme is FALSE or all of the values of the discrete range of a for iteration scheme have been assigned to the iteration parameter. (§ 8.9)

B.42 complete context: A declaration, a specification, or a statement; complete contexts are used in overload resolution. (§ 10.5)

B.43 composite type: A type whose values have elements. There are two classes of composite types: *array types* and *record types*.(§3, § *3.2*)

B.44 concurrent statement: A statement that executes asynchronously,with no defined relative order. Concurrent statements are used for dataflow and structural descriptions. (§9)

B.45 configuration: A construct that defines how component instances in a given block are bound to design entities in order to describe how design entities are put together to form a complete design. (*§*1, § 1.3 , § 5.2)

B.46 conform: Two subprogram specifications, are said to conform if, apart from certain allowed minor variations, both specifications are formed by the same sequence of lexical elements, and corresponding lexical elements are given the same meaning by the visibility rules. Conformance is defined similarly for deferred constant declarations. (§ 2.7)

B.47 connected: A formal port associated with an actual port or signal. A formal port associated with the reserved word **open** is said to be *unconnected*. (§ 1.1.1.2)

B.48 constant: An object whose value may not be changed. Constants maybe *explicitly declared*, subelements of explicitly declared constants, or interface constants. Constants declared in packages may also be *deferred constants*. (§ 4.3.1.1)

B.49 constraint: A subset of the values of a type. The set of possible values for an object of a given type that can be subjected to a condition is called a *constraint*. A value is said to *satisfy* the constraint if it satisfies the corresponding condition. There are index constraints,range constraints, and size constraints. (§3)

B.50 conversion function: A function used to convert values flowing through associations. For interface objects of mode **in**, conversion functions are allowed only on actuals. For interface objects of mode **out** or **buffer**, conversion functions are allowed only on formals. For interface objects of mode **inout** or **linkage**, conversion functions are allowed on both formals and actuals. Conversion functions have a single parameter. A conversion function associated with an actual accepts the type of the actual and returns the type of the formal. A conversion function associated with a formal accepts the type of the formal and returns the type of the actual. (§ 4.3.2.2)

B.51 convertible: A property of an operand with respect to some type. An operand is convertible to some type if there exists an implicit conversion to that type. (§ 7.3.5)

B.52 current value: The value component of the single transaction of a driver whose time component is not greater than the current simulation time.(§ 12.6 , *§* 12.6.1 , § 12.6.2 , § 12.6.3)

B.53 decimal literal: An abstract literal that is expressed in decimal notation. The base of the literal is implicitly 10. The literal may optionally contain an exponent or a decimal point and fractional part.(§ 13.4.1)

B.54 declaration: A construct that defines a declared entity and associates an identifier (or some other notation) with it. This association is in effect within a region of text that is called the *scope* of the declaration. Within the scope of a declaration, there are places where it is possible to use the identifier to refer to the associated declared entity; at such places, the identifier is said to be the *simple name* of the named entity. The simple name is said to *denote* the associated named entity.(§4)

B.55 declarative part: A syntactic component of certain declarations or statements (such as entity declarations, architecture bodies, and block statements). The declarative part defines the lexical area (usually introduced by a keyword such as **is** and terminated with another keyword such as **begin**) within which declarations may occur. (*§* 1.1.2 , *§* 1.2.1 , § 1.3 , § 2.6 , § 9.1 , § 9.2 , § 9.6.1 , § 9.6.2)

B.56 declarative region: A semantic component of certain declarations or statements. A declarative region may include disjoint parts, such as the declarative region of an entity declaration, which extends to the end of any architecture body for that entity. (§ 10.1)

B.57 decorate: To associate a user-defined attribute with a named entity and to define the value of that attribute. (§ 5.1)

B.58 default expression: A default value that is used for a formal generic, port, or parameter if the interface object is unassociated. A default expression is also used to provide an initial value for signals and their drivers. (*§ 4.3.1.2* , *§ 4.3.2.2*)

B.59 deferred constant: A constant that is declared without an assignment symbol (:=) and expression in a package declaration. A corresponding full declaration of the constant must exist in the package body to define the value of the constant. (§ 4.3.1.1)

B.60 delta cycle: A simulation cycle in which the simulation time at the beginning of the cycle is the same as at the end of the cycle. That is,simulation time is not advanced in a delta cycle. Only nonpostponed processes can be executed during a delta cycle. (§ 12.6.4)

B.61 denote: A property of the identifier given in a declaration. Where the declaration is visible, the identifier given in the declaration is said to *denote* the named entity declared in the declaration.(§4)

B.62 depend (on a library unit): A design unit that explicitly or implicitly mentions other library units in a use clause. These dependencies affect the allowed order of analysis of design units. (§ 11.4)

B.63 depend (on a signal value): A property of an implicit signal with respect to some other signal. The current value of an implicit signal R is said to *depend on* the current value of another signal S if R denotes an implicit signal S'Stable(T), S'Quiet(T), or S'Transaction, or if R denotes an implicit GUARD signal and S is any other implicit signal named within the guard expression that defines the current value of R.(§ 12.6.3)

B.64 descending range: A range L **downto** R. (§ 3.1)

B.65 design entity: An entity declaration together with an associated architecture body. Different design entities may share the same entity declaration, thus describing different components with the same interface or different views of the same component. (§1)

B.66 design file: One or more design units in sequence. (§ 11.1)

B.67 design hierarchy: The complete representation of a design that results from the successive decomposition of a design entity into subcomponents and binding of those components to other design entities that may be decomposed in a similar manner. (§1)

B.68 design library: A host-dependent storage facility for intermediate-form representations of analyzed design units. (§ 11.2)

B.69 design unit: A construct that can be independently analyzed and stored in a design library. A design unit may be an entity declaration, an architecture body, a configuration declaration, a package declaration, or a package body declaration. (§ 11.1)

B.70 designate: A property of access values that relates the value to some object when the access value is nonnull. A nonnull access value is said to *designate* an object. (§ 3.3)

B.71 designated subtype: For an access type, the subtype defined by the subtype indication of the access type definition. (§ 3.3)

B.72 designated type: For an access type, the base type of the subtype defined by the subtype

indication of the access type definition. (§ 3.3)

B.73 designator:

a) Syntax that forms part of an association element. A formal designator specifies which formal parameter, port, or generic (or which subelement or slice of a parameter, port, or generic) is to be associated with an actual by the given association element. An actual designator specifies which actual expression, signal, or variable is to be associated with a formal (or subelement or subelements of a formal). An actual designator may also specify that the formal in the given association element is to be left unassociated (with an actual designator of **open**). (§ 4.3.2.2)

b) An identifier, character literal, or operator symbol that defines an alias for some other name. (§ 4.3.3)

c) A simple name that denotes a predefined or user-defined attribute in an attribute name, or a user-defined attribute in an attribute specification.(§ 5.1 , § 6.6)

d) An simple name, character literal, or operator symbol, and possibly a signature, that denotes a named entity in the entity name list of an attribute specification. (§ 5.1)

e) An identifier or operator symbol that defines the name of a subprogram.(§ 2.1)

B.74 directly visible: A visible declaration that is not visible by selection. A declaration is directly visible within its immediate scope,excluding any places where the declaration is hidden. A declaration occurring immediately within the visible part of a package can be made directly visible by means of a use clause. (*§ 10.3* , § 10.4). *See also* **visible**.

B.75 discrete array: A one-dimensional array whose elements are of a discrete type. (§ 7.2.3)

B.76 discrete range: A range whose bounds are of a discrete type.(§ 3.2.1 , § 3.2.1.1)

B.77 discrete type: An enumeration type or an integer type. Each value of a discrete type has a position number that is an integer value. Indexing and iteration rules use values of discrete types. (§ 3.1)

B.78 driver: A container for a projected output waveform of a signal. The value of the signal is a function of the current values of its drivers. Each process that assigns to a given signal implicitly contains a driver for that signal. A signal assignment statement affects only the associated driver(s). (§ 12.4.4 , *§ 12.6.1* , § 12.6.2 , § 12.6.3)

B.79 driving value: The value a signal provides as a source of other signals. (§ 12.6.2)

B.80 effective value: The value obtained by evaluating a reference to the signal within an expression. (§ 12.6.2)

B.81 elaboration: The process by which a declaration achieves its effect. Prior to the completion of its elaboration (including before the elaboration), a declaration is not yet elaborated. (§12)

B.82 element: A constituent of a composite type. (§3) *See also* **subelement**.

B.83 entity declaration: A definition of the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each with the same interface. (*§1*, § 1.1)

B.84 enumeration literal: A literal of an enumeration type. An enumeration literal may be either an identifier or a character literal.(*§* 3.1.1 , § 7.3.1)

B.85 enumeration type: A type whose values are defined by listing(enumerating) them. The values of the type are represented by enumeration literals. (*§ 3.1* , § 3.1.1)

B.86 error: A condition that makes the source description illegal. If an error is detected at the time of analysis of a design unit, it prevents the creation of a library unit for the given design unit. A run-time error causes simulation to terminate. (§ 11.4)

B.87 erroneous: An error condition that cannot always be detected.(§ 2.1.1.1 , § 2.2)

B.88 event: A change in the current value of a signal, which occurs when the signal is updated with its effective value. (§ 12.6.2)

B.89 execute:

a) When first the design hierarchy of a model is elaborated, then its nets are initialized, and finally simulation proceeds with repetitive execution of the simulation cycle, during which processes are executed and nets are updated.

b) When a process performs the actions specified by the algorithm described in its

statement part. (§12, § 12.6)

B.90 expanded name: A selected name (in the syntactic sense) that denotes one or all of the primary units in a library or any named entity within a primary unit. (*§* 6.3 , § 8.1) *See also* **selected name**.

B.91 explicit ancestor: The parent of the implicit signal that is defined by the predefined attributes 'DELAYED, 'QUIET, 'STABLE, or 'TRANSACTION. It is determined using the prefix of the attribute. If the prefix denotes an explicit signal or a slice or subelement (or slice thereof),then that is the explicit ancestor of the implicit signal. If the prefix is one of the implicit signals defined by the predefined attributes 'DELAYED,'QUIET, 'STABLE, or 'TRANSACTION, this rule is applied recursively. If the prefix is an implicit signal GUARD, the signal has no explicit ancestor.(§ 2.2)

B.92 explicit signal: A signal defined by the predefined attributes 'DELAYED, 'QUIET, 'STABLE, or 'TRANSACTION. (§ 2.2)

B.93 explicitly declared constant: A constant of a specified type that is declared by a constant declaration. (§ 4.3.1.1)

B.94 explicitly declared object: An object of a specified type that is declared by an object declaration. An object declaration is called a *single-object declaration* if its identifier list has a single identifier; it is called a *multiple-object declaration* if the identifier list has two or more identifiers. (§ 4.3 , *§* 4.3.1) *See also* **implicitly declared object**.

B.95 expression: A formula that defines the computation of a value.(§ 7.1)

B.96 extend: A property of source text forming a declarative region with disjoint parts. In a declarative region with disjoint parts, if a portion of text is said to *extend* from some specific point of a declarative region to the end of the region, then this portion is the corresponding subset of the declarative region (and does not include intermediate declarative items between an interface declaration and a corresponding body declaration).(§ 10.1)

B.97 extended digit: A lexical element that is either a digit or a letter. (§ 13.4.2)

B.98 external block: A top-level design entity that resides in a library and may be used as a component in other designs. (§1)

B.99 file type: A type that provides access to objects containing a sequence of values of a given type. File types are typically used to access files in the host system environment. The

value of a file object is the sequence of values contained in the host system file. (§3, *§* 3.4)

B.100 floating point types: A discrete scalar type whose values approximate real numbers. The representation of a floating point type includes a minimum of six decimal digits of precision. (§ 3.1 , § 3.1.4)

B.101 foreign subprogram: A subprogram that is decorated with the attribute 'FOREIGN, defined in package STANDARD. The STRING value of the attribute may specify implementation-dependent information about the foreign subprogram. Foreign subprograms may have non-VHDL implementations. An implementation may place restrictions on the allowable modes, classes, and types of the formal parameters to a foreign subprogram, such as constraints on the number and allowable order of the parameters. (§ 2.2)

B.102 formal: A formal port or formal generic of a design entity, a block statement, or a formal parameter of a subprogram. (*§* 2.1.1 , *§* 4.3.2.2 , *§* 5.2.1.2 , § 9.1)

B.103 full declaration: A constant declaration occurring in a package body with the same identifier as that of a deferred constant declaration in the corresponding package declaration. A full type declaration is a type declaration corresponding to an incomplete type declaration. (§ 2.6)

B.104 fully bound: A binding indication for the component instance implies an entity interface and an architecture. (§ 5.2.1.1)

B.105 generate parameter: A constant object whose type is the base type of the discrete range of a generate parameter specification. A generate parameter is declared by a generate statement. (§ 9.7)

B.106 generic: An interface constant declared in the block header of a block statement, a component declaration, or an entity declaration. Generics provide a channel for static information to be communicated to a block from its environment. Unlike constants, however, the value of a generic can be supplied externally, either in a component instantiation statement or in a configuration specification. (§ 1.1.1.1)

B.107 generic interface list: A list that defines local or formal generic constants. (§ 1.1.1.1 , § 4.3.2.1)

B.108 globally static expression: An expression that can be evaluated as soon as the design hierarchy in which it appears is elaborated. A locally static expression is also globally static unless the expression appears in a dynamically elaborated context. (§ 7.4)

B.109 globally static primary: A primary whose value can be determined during the elaboration of its complete context and that does not thereafter change. Globally static primaries can only appear within statically elaborated contexts. (§ 7.4.2)

B.110 group: A named collection of named entities. Groups relate different named entities for the purposes not specified by the language. In particular, groups may be decorated with attributes. (§ 4.6 , § 4.7)

B.111 guard: *See* **guard expression**.

B.112 guard expression: A Boolean-valued expression associated with a block statement that controls assignments to guarded signals within the block. A guard expression defines an implicit signal GUARD that may be used to control the operation of certain statements within the block. (§ 4.3.1.2 , § 9.1 , § 9.5)

B.113 guarded assignment: A concurrent signal assignment statement that includes the option **guarded**, which specifies that the signal assignment statement is executed when a signal GUARD changes from FALSE to TRUE, or when that signal has been TRUE and an event occurs on one of the signals referenced in the corresponding GUARD expression. The signal GUARD may be one of the implicitly declared GUARD signals associated with block statements that have guard expressions, or it may be an explicitly declared signal of type Boolean that is visible at the point of the concurrent signal assignment statement.(§ 9.5)

B.114 guarded signal: A signal declared as a register or a bus. Such signals have special semantics when their drivers are updated from within guarded signal assignment statements. (§ 4.3.1.2)

B.115 guarded target: A signal assignment target consisting only of guarded signals. An unguarded target is a target consisting only of unguarded signals. (§ 9.5)

B.116 hidden: A declaration that is not directly visible. A declaration may be *hidden* in its scope by a homograph of the declaration. (§ 10.3)

B.117 homograph: A reflexive property of two declarations. Each of two declarations is said to be a *homograph* of the other if both declarations have the same identifier and overloading is allowed for at most one of the two. If overloading is allowed for both declarations, then each of the two is a homograph of the other if they have the same identifier, operator symbol, or character literal, as well as the same parameter and result type profile.(§ 1.3.1 , § *10.3*)

B.118 identify: A property of a name appearing in an element association of an assignment

target in the form of an aggregate. The name is said to *identify* a signal or variable and any subelements of that signal or variable. (§ 8.4 , § 8.5)

B.119 immediate scope: A property of a declaration with respect to the declarative region within which the declaration immediately occurs. The immediate scope of the declaration extends from the beginning of the declaration to the end of the declarative region. (§ 10.2)

B.120 immediately within: A property of a declaration with respect to some declarative region. A declaration is said to occur *immediately within* a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any)associated with the declaration itself. (§ 10.1)

B.121 implicit signal: Any signal S'Stable(T), S'Quiet(T), S'Delayed, or S'Transaction, or any implicit GUARD signal. A slice or subelement (or slice thereof) of an implicit signal is also an implicit signal.(§ 12.6.2 , § 12.6.3 , § 12.6.4)

B.122 implicitly declared object: An object whose declaration is not explicit in the source description, but is a consequence of other constructs; for example, signal GUARD. (*§* 4.3 , § 9.1 , § 14.1) *See also* **declared object**.

B.123 imply: A property of a binding indication in a configuration specification with respect to the design entity indicated by the binding specification. The binding indication is said to *imply* the design entity; the design entity maybe indicated directly, indirectly, or by default. (§ 5.2.1.1)

B.124 impure function: A function that may return a different value each time it is called, even when different calls have the same actual parameter values. A pure function returns the same value each time it is called using the same values as actual parameters. An impure function can update objects outside of its scope and can access a broader class of values than a pure function. (§2)

B.125 incomplete type declaration: A type declaration that is used to define mutually dependent and recursive access types. (§ 3.3.1)

B.126 index constraint: A constraint that determines the index range for every index of an array type, and thereby the bounds of the array. An index constraint is *compatible* with an array type if and only if the constraint defined by each discrete range in the index constraint is compatible with the corresponding index subtype in the array type. An array value *satisfies* an index constraint if the array value and the index constraint have the same index range at each index position . (§ 3.1 , *§* 3.2.1.1)

B.127 index range: A multidimensional array has a distinct element for each possible sequence of index values that can be formed by selecting one value for each index (in the given order). The possible values for a given index are all the values that belong to the corresponding range. This range of values is called the *index range*. (§ 3.2.1)

B.128 index subtype: For a given index position of an array, the *index subtype* is denoted by the type mark of the corresponding index subtype definition. (§ 3.2.1)

B.129 inertial delay: A delay model used for switching circuits; a pulse whose duration is shorter than the switching time of the circuit will not be transmitted. Inertial delay is the default delay mode for signal assignment statements. (§ 8.4) *See also* **transport delay**.

B.130 initial value expression: An expression that specifies the initial value to be assigned to a variable. (§ 4.3.1.3)

B.131 inputs: The signals identified by the longest static prefix of each signal name appearing as a primary in each expression (other than time expressions) within a concurrent signal assignment statement. (§ 9.5)

B.132 instance: A subcomponent of a design entity whose prototype is a component declaration, design entity, or configuration declaration. Each instance of a component may have different actuals associated with its local ports and generics. A component instantiation statement whose instantiated unit denotes a component creates an instance of the corresponding component. A component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration creates an instance of the denoted design entity. (§ 9.6 , § 9.6.1 , § 9.6.2)

B.133 integer literal: An abstract literal of the type *universal_integer* that does not contain a base point.(§ 13.4)

B.134 integer type: A discrete scalar type whose values represent integer numbers within a specified range. (§ 3.1 , § 3.1.2)

B.135 interface list: A list that declares the interface objects required by a subprogram, component, design entity, or block statement.(§ 4.3.2.1)

B.136 internal block: A nested block in a design unit, as defined by a block statement. (§1)

B.137 ISO: The International Organization for Standardization.

B.138 ISO 8859-1: The ISO Latin-1 character set. Package Standard contains the definition of type Character, which represents the ISO Latin-1character set. (§ 3.1.1 , § 14.2)

B.139 kernel process: A conceptual representation of the agent that coordinates the activity of user-defined processes during a simulation. The kernel process causes the execution of I/O operations, the propagation of signal values, and the updating of values of implicit signals [such as S'Stable(T)]; in addition, it detects events that occur and causes the appropriate processes to execute in response to those events. (§ 12.6)

B.140 left of: When both a value V1 and a value V2 belong to a range and either the range is an ascending range and V2 is the successor of V1, or the range is a descending range and V2 is the predecessor of V1. (§ 3.1)

B.141 left-to-right order: When each value in a list of values is to the left of the next value in the list within that range, except for the last value in the list. (§ 3.1)

B.142 library: *See* **design library**.

B.143 library unit: The representation in a design library of an analyzed design unit. (§ 11.1)

B.144 literal: A value that is directly specified in the description of a design. A literal can be a bit string literal, enumeration literal, numeric literal, string literal, or the literal **null**. (§ 7.3.1)

B.145 local generic: An interface object declared in a component declaration that serves to connect a formal generic in the interface list of an entity and an actual generic or value in the design unit instantiating that entity. (§ 4.3 , § 4.3.2.2 , § 4.5)

B.146 local port: A signal declared in the interface list of a component declaration that serves to connect a formal port in the interface list of an entity and an actual port or signal in the design unit instantiating that entity. (§ 4.3 , § 4.3.2.2 , § 4.5)

B.147 locally static expression: An expression that can be evaluated during the analysis of the design unit in which it appears. (*§ 7.4* , § 7.4.1)

B.148 locally static name: A name in which every expression is locally static (if every discrete range that appears as part of the name denotes a locally static range or subtype and if no prefix within the name is either an object or value of an access type or a function call). (§ 6.1)

B.149 locally static primary: One of a certain group of primaries that includes literals, certain

constants, and certain attributes. (§ 7.4)

B.150 locally static subtype: A subtype whose bounds and direction can be determined during the analysis of the design unit in which it appears.(§ 7.4.1)

B.151 longest static prefix: The name of a signal or a variable name,if the name is a static signal or variable name. Otherwise, the longest static prefix is the longest prefix of the name that is a static signal or variable name. (§ 6.1) *See also* **static signal name**.

B.152 loop parameter: A constant, implicitly declared by the for clause of a loop statement, used to count the number of iterations of a loop.(§ 8.9)

B.153 lower bound: For a range L **to** R or L **downto** R, the smaller of L and R. (§ 3.1)

B.154 match: A property of a signature with respect to the parameter and subtype profile of a subprogram or enumeration literal. The signature is said to *match* the parameter and result type profile if certain conditions are true. (§ 2.3.2)

B.155 matching elements: Corresponding elements of two composite type values that are used for certain logical and relational operations.(§ 7.2.3)

B.156 member: A slice of an object, a subelement, or an object; or a slice of a subelement of an object. (§ 3)

B.157 mode: The direction of information flow through the port or parameter. Modes are **in**, **out**, **inout**, **buffer**, or **linkage**. (§ 4.3.2)

B.158 model: The result of the elaboration of a design hierarchy. The *model* can be executed in order to simulate the design it represents.(§12, § 12.6)

B.159 name: A property of an identifier with respect to some named entity. Each form of declaration associates an identifier with a named entity. In certain places within the scope of a declaration, it is valid to use the identifier to refer to the associated named entity; these places are defined by the visibility rules. At such places, the identifier is said to be the *name* of the named entity. (*§*4, § 6.1)

B.160 named association: An association element in which the formal designator appears explicitly. (§ 4.3.2.2 , § 7.3.2)

B.161 named entity: An item associated with an identifier, character literal, or operator

symbol as the result of an explicit or implicit declaration. (§4) *See also* **name**.

B.162 net: A collection of drivers, signals (including ports and implicit signals), conversion functions, and resolution functions that connect different processes. Initialization of a net occurs after elaboration, and a net is updated during each simulation cycle. (§12, § 12.1 , *§* 12.6.2)

B.163 nonobject alias: An alias whose designator denotes some named entity other than an object. (*§* 4.3.3 , § 4.3.3.2) *See also* **object alias**.

B.164 nonpostponed process: An explicit or implicit process whose source statment does not contain the reserved word **postponed.** When a nonpostponed process is resumed, it executes in the current simulation cycle. Thus, nonpostponed processes have access to the current values of signals,whether or not those values are stable at the current model time. (§ 9.2)

B.165 null array: Any of the discrete ranges in the index constraint of an array that define a null range. (§ 3.2.1.1)

B.166 null range: A range that specifies an empty subset of values. A range L **to** R is a null range if L > R, and range L **downto** R is a null range if L < R. (§ 3.1)

B.167 null slice: A slice whose discrete range is a null range.(§ 6.5)

B.168 null waveform element: A waveform element that is used to turn off a driver of a guarded signal. (§ 8.4.1)

B.169 null transaction: A transaction produced by evaluating a null waveform element. (§ 8.4.1)

B.170 numeric literal: An abstract literal, or a literal of a physical type. (§ 7.3.1)

B.171 numeric type: An integer type, a floating point type, or a physical type. (§ 3.1)

B.172 object: A named entity that has a value of a given type. An object can be a constant, signal, variable, or file. (§ 4.3.3)

B.173 object alias: An alias whose alias designator denotes an object(that is, a constant, signal, variable, or file). (*§* 4.3.3 , *§;* 4.3.3.1) *See also* **nonobject alias**.

B.174 overloaded: Identifiers or enumeration literals that denote two different named entities. Enumeration literals, subprograms, and predefined operators may be overloaded. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal must be determinable from the context. (§ 2.1 , § 2.3 , § 2.3.1 , § 2.3.2 , § 3.1.1)

B.175 parameter: A constant, signal, variable, or file declared in the interface list of a subprogram specification. The characteristics of the class of objects to which a given parameter belongs are also characteristics of the parameter. In addition, a parameter has an associated mode that specifies the direction of data flow allowed through the parameter. (§ 2.1.1 , § 2.1.1.1 , § 2.1.1.2 , § 2.1.1.3 , § 2.3 , § 2.6)

B.176 parameter interface list: An interface list that declares the parameters for a subprogram. It may contain interface constant declarations,interface signal declarations, interface variable declarations, interface file declarations, or any combination thereof. (§ 4.3.2.1)

B.177 parameter type profile: Two formal parameter lists that have the same number of parameters, and at each parameter position the corresponding parameters have the same base type. (§ 2.3)

B.178 parameter and result type profile: Two subprograms that have the same parameter type profile, and either both are functions with the same result base type, or neither of the two is a function. (§ 2.3)

B.179 parent: A process or a subprogram that contains a procedure call statement for a given procedure or for a parent of the given procedure.(§ 2.2)

B.180 passive process: A process statement where neither the process itself, nor any procedure of which the process is a parent, contains a signal assignment statement. (§ 9.2)

B.181 physical literal: A numeric literal of a physical type.(§ 3.1.3)

B.182 physical type: A numeric scalar type that is used to represent measurements of some quantity. Each value of a physical type has a position number that is an integer value. Any value of a physical type is an integral multiple of the primary unit of measurement for that type. (§ 3.1 , § 3.1.3)

B.183 port: A channel for dynamic communication between a block and its environment. A signal declared in the interface list of an entity declaration,in the header of a block statement,

or in the interface list of a component declaration. In addition to the characteristics of signals, ports also have an associated mode; the mode constrains the directions of data flow allowed through the port. (§ *1.1.1.2* , § 4.3.1.2)

B.184 port interface list: An interface list that declares the inputs and outputs of a block, component, or design entity. It consists entirely of interface signal declarations. (§ 1.1.1 , § *1.1.1.2* , § 4.3.2.1 , § 4.3.2.2 , § 9.1)

B.185 positional association: An association element that does not contain an explicit appearance of the formal designator. An actual designator at a given position in an association list corresponds to the interface element at the same position in the interface list. (§ 4.3.2.2 , § 7.3.2)

B.186 postponed process: An explicit or implicit process whose source statement contains the reserved word **postponed.** When a postponed process is resumed, it does not execute until the final simulation cycle at the current modeled time. Thus, a postponed process accesses the values of signals that are the "stable" values at the current simulated time. (§ 9.2)

B.187 predefined operators: Implicitly defined operators that operate on the predefined types. Every predefined operator is a pure function. No predefined operators have named formal parameters; therefore, named association may not be used when invoking a predefined operation. (§ 7.2 , § 14.2)

B.188 primary: One of the elements making up an expression. Each primary has a value and a type. (§ 7.1)

B.189 projected output waveform: A sequence of one or more transactions representing the current and projected future values of the driver.(§ 12.6.1)

B.190 pulse rejection limit: The threshold time limit for which a signal value whose duration is greater than the limit will be propagated. A pulse rejection limit is specified by the reserved word **reject** in an inertially delayed signal assignment statement. (§ 8.4)

B.191 pure function: A function that returns the same value each time it is called with the same values as actual parameters. An *impure* function may return a different value each time it is called, even when different calls have the same actual parameter values. (§ 2.1)

B.192 quiet: In a given simulation cycle, a signal that is not active.(§ 12.6.2)

B.193 range: A specified subset of values of a scalar type. (§ 3.1)*See also* **ascending range**,

belong (to a range),**descending range**, **lower bound**, and **upper bound**.

B.194 range constraint: A construct that specifies the range of values in a type. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype or if the range constraint defines a null range. The direction of a range constraint is the same as the direction of its range. (*§* 3.1 , *§* 3.1.2 , § 3.1.3 , § 3.1.4)

B.195 read: The value of an object is said to be *read* when its value is referenced or when certain of its attributes are referenced.(§ 4.3.2)

B.196 real literal: An abstract literal of the type *universal_real* that contains a base point. (§ 13.4)

B.197 record type: A composite type whose values consist of named elements. (*§* 3.2.2 , § 7.3.2.1)

B.198 reference: Access to a named entity. Every appearance of a designator (a name, character literal, or operator symbol) is a reference to the named entity denoted by the designator, unless the designator appears in a library clause or use clause. (§ 10.4 , § 11.2)

B.199 register: A kind of guarded signal that retains its last driven value when all of its drivers are turned off. (§ 4.3.1.2)

B.200 regular structure: Instances of one or more components arranged and interconnected (via signals) in a repetitive way. Each instance may have characteristics that depend upon its position within the group of instances. Regular structures may be represented through the use of the generate statement. (§ 9.7)

B.201 resolution: The process of determining the resolved value of a resolved signal based on the values of multiple sources for that signal.(§ 2.4 , § 4.3.1.2)

B.202 resolution function: A user-defined function that computes the resolved value of a resolved signal. (§ 2.4 , § 4.3.1.2)

B.203 resolution limit: The primary unit of type TIME (by default, 1femtosecond). Any TIME value whose absolute value is smaller than this limit is truncated to zero (0) time units. (§ 3.1.3.1)

B.204 resolved signal: A signal that has an associated resolution function. (§ 4.3.1.2)

B.205 resolved value: The output of the resolution function associated with the resolved signal, which is determined as a function of the collection of inputs from the multiple sources of the signal. (*§* 2.4 , § 4.3.1.2)

B.206 resource library: A library containing library units that are referenced within the design unit being analyzed. (§ 11.2)

B.207 result subtype: The subtype of the returned value of a function.(§ 2.1)

B.208 resume: The action of a wait statement upon an enclosing process when the conditions on which the wait statement is waiting are satisfied. If the enclosing process is a nonpostponed process, the process will subsequently execute during the current simulation cycle. Otherwise, the process is a postponed process, which will execute during the final simulation cycle at the current simulated time. (§ 12.6.3)

B.209 right of: When a value V1 and a value V2 belong to a range and either the range is an ascending range and V2 is the predecessor of V1, or the range is a descending range and V2 is the successor of V1. (§ 14.1)

B.210 satisfy: A property of a value with respect to some constraint. The value is said to *satisfy* a constraint if the value is in the subset of values determined by the constraint. (*§*3, *§* 3.2.1.1)

B.211 scalar type: A type whose values have no elements. Scalar types consist of *enumeration types*, *integer types*, *physical types*, and *floating point types*. Enumeration types and integer types are called *discrete types*. Integer types, floating point types,and physical types are called *numeric types*. All scalar types are ordered; that is, all relational operators are predefined for their values.(§3, § *3.1*)

B.212 scope: A portion of the text in which a declaration may be visible. This portion is defined by visibility and overloading rules.(§ 10.2)

B.213 selected name: Syntactically, a name having a prefix and suffix separated by a dot. Certain selected names are used to denote record elements or objects denoted by an access value. The remaining selected names are referred to as *expanded names*. (*§* 6.3 , § 8.1) Also see **expanded name**.

B.214 sensitivity set: The set of signals to which a wait statement is sensitive. The sensitivity set is given explicitly in an**on** clause, or is implied by an **until** clause. (§ 8.1)

B.215 sequential statements: Statements that execute in sequence in the order in which they appear. Sequential statements are used for algorithmic descriptions. (§8)

B.216 short-circuit operation: An operation for which the right operand is evaluated only if the left operand has a certain value. The short-circuit operations are the predefined logical operations **and**, **or**,**nand**, and **nor** for operands of types BIT and BOOLEAN.(§ 7.2)

B.217 signal: An object with a past history of values. A signal may have multiple drivers, each with a current value and projected future values. The term *signal* refers to objects declared by signal declarations or port declarations. (§ 4.3.1.2)

B.218 signal transform: A sequential statement within a statement transform that determines which one of the alternative waveforms, if any, is to be assigned to an output signal. A signal transform can be a sequential signal assignment statement, an if statement, a case statement, or a null statement.(§ 9.5)

B.219 simple name: The identifier associated with a named entity, either in its own declaration or in an alias declaration. (§6.2)

B.220 simulation cycle: One iteration in the repetitive execution of the processes defined by process statements in a model. The first simulation cycle occurs after initialization. A simulation cycle can be a delta cycle or a time-advance cycle. (§ 12.6.4)

B.221 single-object declaration: An object declaration whose identifier list contains a single identifier; it is called a multiple-object declaration if the identifier list contains two or more identifiers. (§ 4.3.1)

B.222 slice: A one-dimensional array of a sequence of consecutive elements of another one-dimensional array. (§ 6.5)

B.223 source: A contributor to the value of a signal. A source can be a driver or port of a block with which a signal is associated or a composite collection of sources. (§ 4.3.1.2)

B.224 specification: A class of construct that associates additional information with a named entity. There are three kinds of specifications:attribute specifications, configuration specifications, and disconnection specifications. (§5)

B.225 statement transform: The first sequential statement in the process equivalent to the concurrent signal assignment statement. The statement transform defines the actions of the concurrent signal assignment statement when it executes. The statement transform is followed

by a wait statement, which is the final statement in the equivalent process. (§ 9.5)

B.226 static: *See* **locally static** and **globally static**.

B.227 static name: A name in which every expression that appears as part of the name (for example, as an index expression) is a static expression (if every discrete range that appears as part of the name denotes a static range or subtype and if no prefix within the name is either an object or value of an access type or a function call). (§ 6.1)

B.228 static range: A range whose bounds are static expressions.(§ 7.4)

B.229 static signal name: A static name that denotes a signal.(§ 6.1)

B.230 static variable name: A static name that denotes a variable.(§ 6.1)

B.231 string literal: A sequence of graphic characters, or possibly none, enclosed between two quotation marks ("). The type of a string literal is determined from the context. (§ 7.3.1 , § 13.6)

B.232 subaggregate: An aggregate appearing as the expression in an element association within another, multidimensional array aggregate. The subaggregate is an (*n*-1)-dimensional array aggregate, where *n* is the dimensionality of the outer aggregate. Aggregates of multidimensional arrays are expressed in row-major (rightmost index varies fastest) order.(§ 7.3.2.2)

B.233 subelement: An element of another element. Where other subelements are excluded, the term *element* is used. (§3)

B.234 subprogram specification: Specifies the designator of the subprogram, any formal parameters of the subprogram, and the result type for a function subprogram. (§ 2.1)

B.235 subtype: A type together with a constraint. A value *belongs to* a subtype of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself. Such a subtype is said to be *unconstrained* because it corresponds to a condition that imposes no restriction. (§3)

B.236 suspend: A process that stops executing and waits for an event or for a time period to elapse. (§ 12.6.4)

B.237 timeout interval: The maximum time a process will be suspended, as specified by the

timeout period in the **until** clause of a wait statement.(§ 8.1)

B.238 to the left of: *See* **left of**.

B.239 to the right of: *See* **right of**.

B.240 transaction: A pair consisting of a value and a time. The value represents a (current or) future value of the driver; the time represents the relative delay before the value becomes the current value. (§ 12.6.1)

B.241 transport delay: An optional delay model for signal assignment. Transport delay is characteristic of hardware devices (such as transmission lines) that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. (§ 8.4) *See also* **inertial delay**.

B.242 type: A set of values and a set of operations. (§3)

B.243 type conversion: An expression that converts the value of a subexpression from one type to the designated type of the type conversion. Associations in the form of a type conversion are also allowed. These associations have functions and restrictions similar to conversion functions but can be used in places where conversion functions cannot. In both cases(expressions and associations), the converted type must be closely related to the designated type. (§ 4.3.2.2 , § 7.3.5) *See also* **conversion function** and **closely related types**.

B.244 unaffected: A waveform in a concurrent signal assignment statement that does not affect the driver of the target. (*§ 8.4* , § 9.5.1)

B.245 unassociated formal: A formal that is not associated with an actual. (§ 5.2.1.2)

B.246 unconstrained subtype: A subtype that corresponds to a condition that imposes no restriction. (*§3*, § 4.2)

B.247 unit name: A name defined by a unit declaration (either the primary unit declaration or a secondary unit declaration) in a physical type declaration. (§ 3.1.3)

B.248 universal_integer: An anonymous predefined integer type that is used for all integer literals. The position number of an integer value is the corresponding value of the type *universal_integer*. (*§* 3.1.2 , § 7.3.1 , § 7.3.5)

B.249 universal_real: An anonymous predefined type that is used for literals of floating point

types. Other floating point types have no literals. However, for each floating point type there exists an implicit conversion that converts a value of type *universal_real* into the corresponding value (if any) of the floating point type. (§ 3.1.4 , § 7.3.1 , § 7.3.5)

B.250 update: An action on the value of a signal, variable, or file. The value of a signal is said to be *updated* when the signal appears as the target (or a component of the target) of a signal assignment statement,(indirectly) when it is associated with an interface object of mode **out**,**buffer**, **inout**, or **linkage**, or when one of its subelements(individually or as part of a slice) is updated. The value of a signal is also said to be *updated* when it is subelement or slice of a resolved signal,and the resolved signal is updated. The value of a variable is said to be *updated* when the variable appears as the target (or a component of the target) of a variable assignment statement, (indirectly) when it is associated with an interface object of mode **out** or **linkage**, or when one of its subelements (individually or part of a slice) is updated. The value of a file is said to be *updated* when a WRITE operation is performed on the file object. (§ 4.3.2)

B.251 upper bound: For a range L **to** R or L **downto** R, the larger of L and R. (§ 3.1)

B.252 variable: An object with a single current value.(§ 4.3.1.3)

B.253 visible: When the declaration of an identifier defines a possible meaning of an occurrence of the identifier used in the declaration. A visible declaration is visible by selection (for example, by using an expanded name) or directly visible (for example, by using a simple name). (§ 10.3)

B.254 waveform: A series of transactions, each of which represents a future value of the driver of a signal. The transactions in a waveform are ordered with respect to time, so that one transaction appears before another if the first represents a value that will occur sooner than the value represented by the other. (§ 8.4)

B.255 white space character: A space, a nonbreaking space, or a horizontal tabulation character (SP, NBSP, or HT). (§ 14.3)

B.256 working library: A design library into which the library unit resulting from the analysis of a design unit is placed. (§ 11.2)

Annex C

Potentially nonportable constructs (informative)

This annex lists those VHDL constructs whose use may result in nonportable descriptions. A description is considered portable if it

a) Compiles, elaborates, initializes, and simulates to termination of the simulation cycle on all conformant implementations, and

b) The time-variant state of all signals and variables in the description are the same at all times during the simulation

under the condition that the same stimuli are applied at the same times to the description. The stimuli applied to a model include the values supplied to generics and ports at the root of the design hierarchy of the model, if any.

Note that the content of files generated by a description are not part of the state of the description, but that the content of files consumed by a description are part of the state of the description.

The use of the following constructs may lead to nonportable VHDL descriptions:

-- Resolution functions that do not treat all inputs symmetrically.

-- The comparison of floating point values.

-- Events on floating-point-valued signals.

-- The use of explicit type conversion to convert floating point values to integer values.

-- Any value that does not fall within the minimum guaranteed range for the type.

-- The use of architectures and subprogram bodies implemented via the foreign language interface (the 'FOREIGN attribute).

-- Processes that communicate via file I/O, including TEXTIO.

-- Impure functions.

-- Linkage ports.

-- Ports and generics in the root of a design hierarchy.

-- Use of a time resolution greater than fs.

-- Shared variables.

-- Procedure calls passing a single object of an array or record type to multiple formals where at least one of the formals is of mode **out** or **inout**.

-- Models that depend on a particular format of T'Image.

-- Declarations of integer or physical types that have a secondary unit whose position number is outside of the range -(2**31-1) **to** 2**31-1.

-- The predefined attributes 'INSTANCE_NAME or 'PATH_NAME, if the behavior of the model is dependent on the values returned by the attributes.

Annex D

Changes from IEEE Std 1076-1987

(informative)

This annex lists those clauses that have been changed from IEEE Std 1076-1987during its revision. The clause numbers are from IEEE Std 1076-1987; where a new clause has been added, it is described as being added between or after existing clauses from IEEE Std 1076-1987.

Section 1: 1.1 , 1.1.1 , 1.1.1.1 , 1.1.1.2 , 1.1.2 , 1.1.3 , 1.2 , 1.2.1 , 1.2.2 , 1.3 , 1.3.1 , and 1.3.2 .

Section 2: 2.1 , 2.1.1 , 2.1.1.1 , 2.1.1.2 , 2.2 , 2.3 , 2.3.1 , 2.4 , 2.5 , 2.6 , and 2.7 . In addition, the following new clauses have been added: 2.1.1.3 ,describing file parameters, has been added after 2.1.1.2 ; and 2.3.2 , describing signatures, has been added after 2.3.1 .

Section 3: Introduction, 3.1 , 3.1.1 , 3.1.1.1 , 3.1.3 , 3.1.3.1 , 3.1.4 , 3.1.4.1 , 3.2.1 , 3.2.1.1 , 3.2.2 , 3.3 , 3.3.1 , and 3.4.1 .

Section 4: Introduction, 4.1 , 4.2 , 4.3 , 4.3.1 , 4.3.1.1 , 4.3.1.2 , 4.3.1.3 , 4.3.2 , 4.3.3 , 4.3.3.1 , 4.3.3.2 , 4.3.4 , 4.4 , and 4.5 . In addition, two new clauses, describing group template declarations and group declarations,have been added at the end of this chapter.

Section 5: Introduction, 5.1 , 5.2 , 5.2.1 , 5.2.1.1 , 5.2.1.2 , 5.2.2 , and 5.3 .

Section 6: 6.1 , 6.2 , 6.3 , 6.4 , 6.5 , and 6.6 .

Section 7: 7.1 , 7.2 , 7.2.1 , 7.2.2 , 7.2.3 , 7.2.4 , 7.3.1 , 7.3.2 , 7.3.2.1 , 7.3.2.2 , 7.3.3 , 7.3.5 , 7.3.6 , 7.4 , and 7.5 . Additionally, a new clause describing the shift operators has been added between 7.2.1 and 7.2.2 , and anew clause describing the sign operators has been added between 7.2.3 and 7.2.4 .

Section 8: Introduction, 8.1 , 8.2 , 8.3 , 8.3.1 , 8.4 , 8.4.1 , 8.5 , 8.6 , 8.7 , 8.8 , 8.9 , 8.10 , 8.11 , and 8.12 . Additionally, a new clause describing the report statement has been added between 8.2 and 8.3 .

Section 9: Introduction, 9.1 , 9.2 , 9.3 , 9.4 , 9.5 , 9.5.1 , 9.5.2 , 9.6 , 9.6.1 , and 9.7 . In addition, a new clause describing the instantiation of a design entity has been added between 9.6.1 and 9.7 .

Section 10: 10.1 , 10.2 , 10.3 , 10.4 , and 10.5 .

Section 11: 11.2 , 11.3 , and 11.4 .

Section 12: Introduction, 12.1 , 12.2 , 12.2.1 , 12.2.2 , 12.2.4 , 12.3 , 12.3.1 , 12.3.1.2 , 12.3.1.3 , 12.3.1.4 , 12.3.1.5 , 12.3.2.1 , 12.4 , 12.4.1 , 12.4.2 , 12.4.3 , 12.6.1 , 12.6.2 , and 12.6.3 .

Section 13: 13.1 , 13.3 , 13.4.2 , 13.5 , and 13.9 . In addition, two new clauses describing basic and extended identifiers have been added between 13.3 and 13.4 .

Section 14: 14.1 , 14.2 , and 14.3 .

Annex E

Related standards

(informative)

ANSI/ISO/IEC 8652-1995 Information Technology - Programming Languages - Ada (revised ANSI/MIL-STD-1815A-1983, American National Standard Reference Manual for the Ada Programming Language.) [1,3]

IEEE Std 1029.1 -1991, IEEE Standard for Waveform and Vector Exchange (WAVES).[2]

IEEE Std 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).

ANSI/ISO 8859-1 : 1987, Information processing--8-bit single-byte coded graphic character sets--Part 1: Latin Alphabet No. 1.[2,3]

[1]ANSI publications are available from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway,NJ 08855-1331, USA.

[3]ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

IEEE VHDL LRM

VHDL HOME PAGE

Basic VHDL - Module 1

This module was prepared as part of the RASSP Education & Facilitation effort.

Version 1.0

Toolbar Functionality

	Takes the user up one hierarchical level in the presentation.
	Takes the user to the previous section of the presentation.
	Takes the user to the previous slide in the presentation.
	Takes the user to a listing of all slides with links to each slide.
	Takes the user to a visual representation of the organization of the slide presentation.
	Takes the user to a document, further explaining the information contained within the slide.
	Brings the user to this document, containing information on the use of the toolbar.
	Takes the user to the next slide in the presentation.
	Takes the user to the next section of the presentation.
	Takes the user down one hierarchical level in the presentation.

Basic VHDL - Module 1

Outline

Introduction - The Need for Education

- **In a survey of 71 US universities (representing about half of the EE graduating seniors in 1993), they reported**
  - **44% have no training on or use of VHDL in any undergraduate EE course**
  - **45% have no faculty members who can teach VHDL**
  - **14% of the graduating seniors have a working knowledge of VHDL and only 8% know Verilog**
- **However, in the 1994 USE/DA Standards Survey, 85% of the engineers surveyed were designers and reported**
  - **55% were familiar with EDIF**
  - **55% were familiar with VHDL**
  - **33% were familiar with Verilog**

Basic VHDL - Module 1

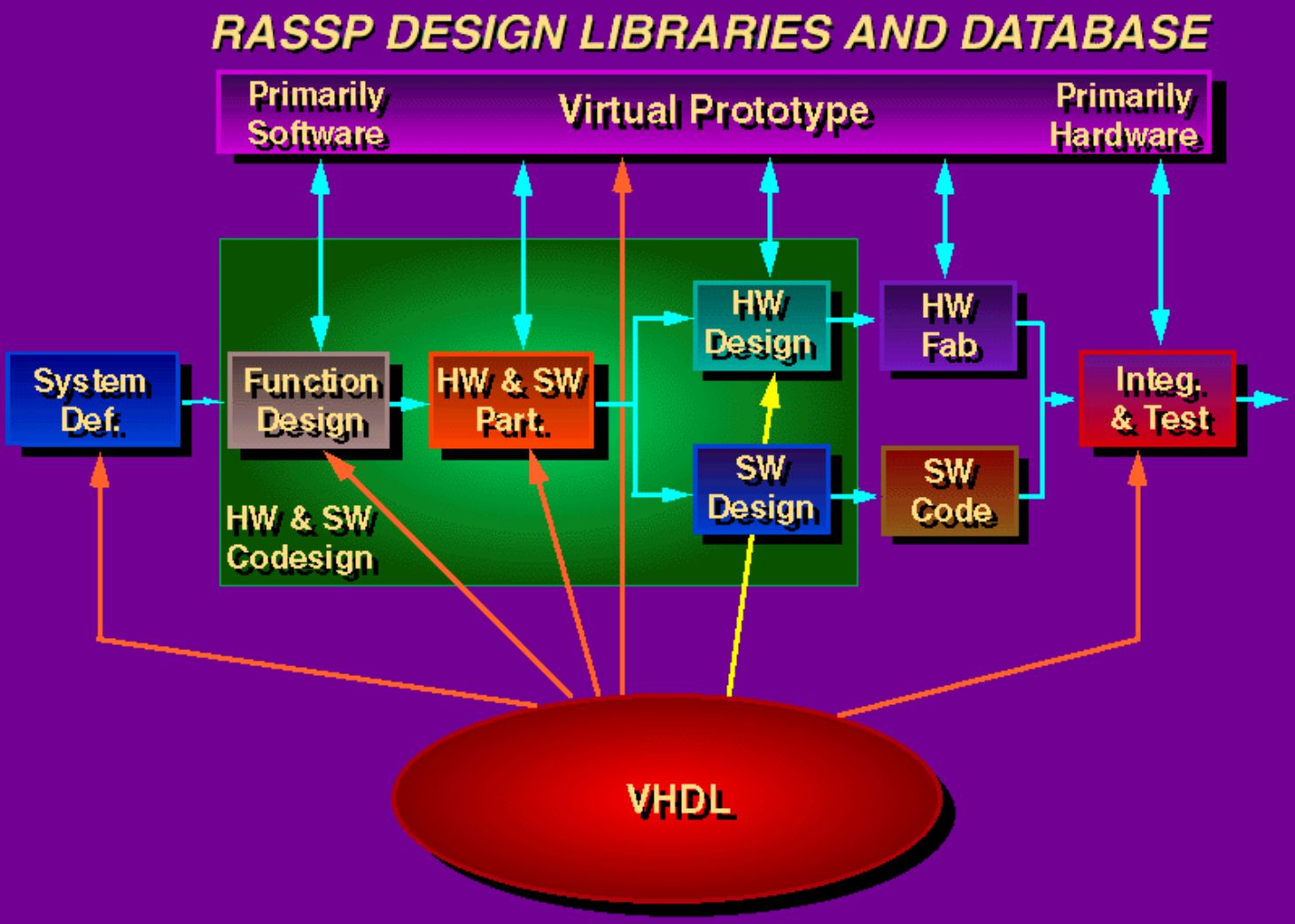
Module Goals

- Comprehension of VHDL Basic Constructs
- Understanding of the VHDL Timing Model
- Familiarity with VHDL design descriptions

IEEE VHDL LRM

RASSP Roadmap

VHDL HOME PAGE



IEEE VHDL LRM

Module Goals

-- Notes Page --

VHDL HOME PAGE

The goals of this module are to provide an introduction to the basic concepts and constructs of VHDL. VHDL is a versatile hardware description language which is useful for modeling electronic systems at various levels of design abstractions. Although most of the language will be touched on in this module, subsequent modules will cover specific areas of VHDL more thoroughly.

Specifically, areas to be covered in this module include:

- The VHDL timing model
- VHDL entities, architectures, and packages
- Concurrent and sequential modes of execution

The goal of this module is to provide a basic understanding of VHDL fundamentals in preparation for the material to be covered in the subsequent VHDL modules.

Need for Education

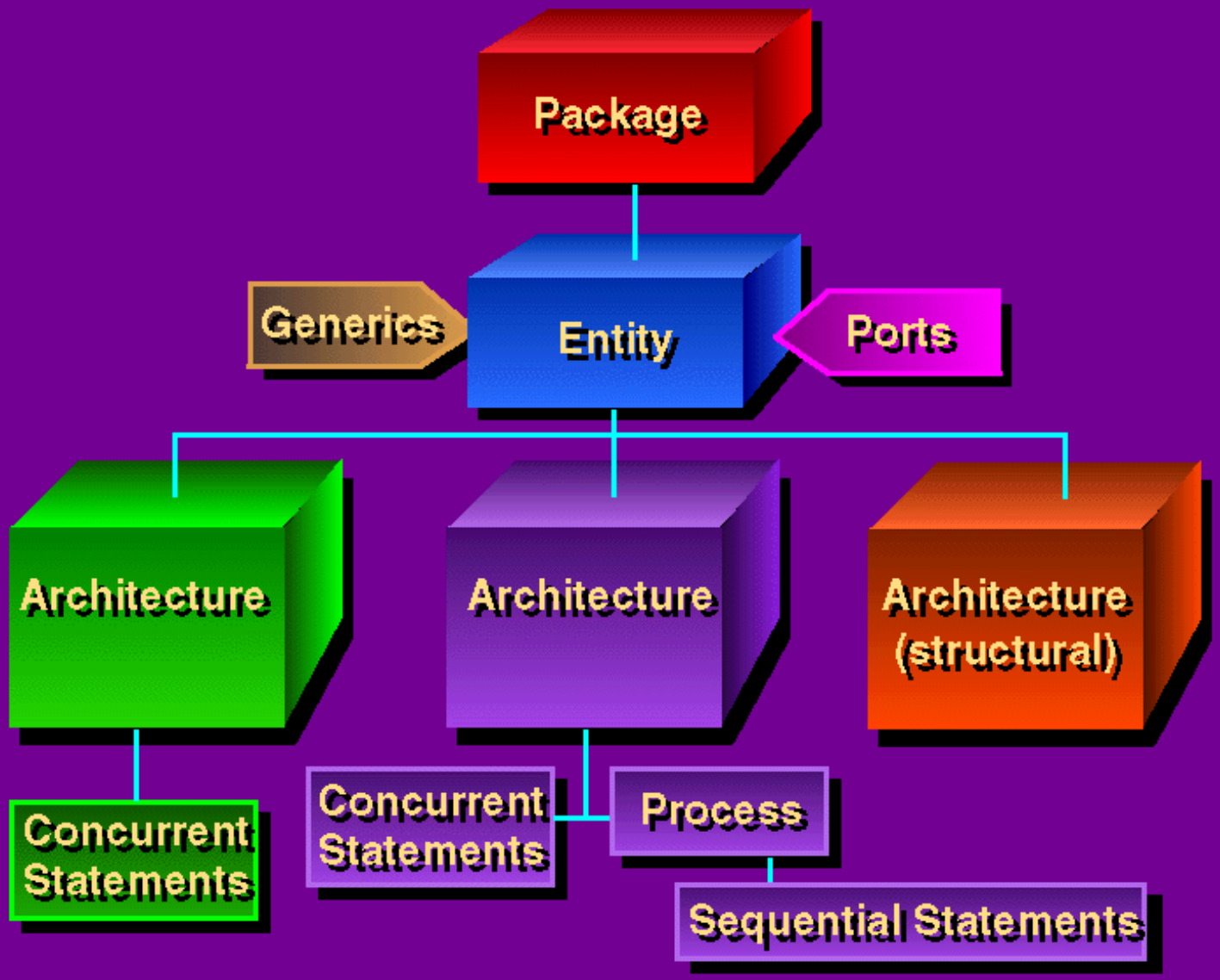
-- Notes Page --

VHDL is an IEEE and U.S. Department of Defense standard for electronic system descriptions. It is also becoming increasingly popular in private industry as experience with the language grows and supporting tools become more widely available. Therefore, to facilitate the transfer of system description information, an understanding of VHDL will become increasingly important. This module provides a first step towards developing a basic comprehension of VHDL.

IEEE VHDL LRM

Putting It All Together

VHDL HOME PAGE



Map Notes Help

Putting it all Together

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This figure captures the main features of a complete VHDL model. A single component model is composed of one entity and one or many architectures. The entity represents the interface specification (I/O) of the component. It defines the components external view, sometimes referred to as its "pins".

The architecture(s) describe the function or composition of an entity. There are three general types of architectures. One type of architecture describes the structure of the design (right hand side) in terms of its sub-components and their interconnections. A key item of a structural VHDL architecture is the "configuration statement" which binds the entity of a sub-component to one of several alternative architectures for that component.

A second type of architecture, containing only concurrent statements, is commonly referred to as a dataflow description (left hand side). Concurrent statements execute when data is available on their inputs. These statements can occur in any order within the architecture.

The third type of architecture is the behavioral description in which the functional and possibly timing characteristics are described using VHDL concurrent statements and processes. The process is a concurrent statement of an architecture. All statements contained within a process execute in a sequential order until it gets suspended by a wait statement.

Packages are used to provide a collection of common declarations,

constants, and/or subprograms to entities and architectures.

Generics provide a method to communicate static information to a architecture from the external environment. They are passed through the entity construct.

Ports provide the mechanism for a device to communication with its environment. A port declaration defines the names, types, directions, and possible default values for the signals in a component's interface.

Implicit in this figure is the testbench which is the top level of a self-contained simulatable model. The testbench is a special VHDL object for which the entity has no signals in its port declaration. Its architecture often contains constructs from all three of the types described above. Structural VHDL concepts are used to connect the model's various components together, Dataflow and behavioral concepts are often used to provide the simulation's start/stop conditions, or other desired modeling directives.

This slide will be used again at the end of this module as a review.

Concepts and History of VHDL

IEEE VHDL LRM

VHDL HOME PAGE

- VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
- VHDL is an international standard specification language for describing digital hardware used by industry worldwide
- VHDL enables hardware modeling from the gate to system level
- VHDL provides a mechanism for digital design and reusable design documentation

IEEE VHDL LRM

History of VHDL

VHDL HOME PAGE

- **Very High Speed Integrated Circuit (VHSIC) Program**
  - **Launched in 1980**
  - **Aggressive effort to advance state of the art**
  - **Object was to achieve significant gains in VLSI technology**
  - **Need for common descriptive language**
  - **$ 17 Million for direct VHDL development**
  - **$ 16 Million for VHDL design tools**
- **Woods Hole Workshop**
  - **Held in June 1981 in Massachusetts**
  - **Discussion of VHSIC goals**
  - **Comprised of members of industry, government, and academia**
- **In July 1983, a team of Intermetrics, IBM and Texas Instruments were awarded a contract to develop VHDL**
- **In August 1985, the final version of the language under government contract was released: VHDL Version 7.2**
- **In December 1987, VHDL became IEEE Standard 1076-1987 and in 1988 an ANSI approved standard**
- **In September 1993, VHDL was restandardized to**

clarify and enhance the language (IEEE Standard 1076-1993)

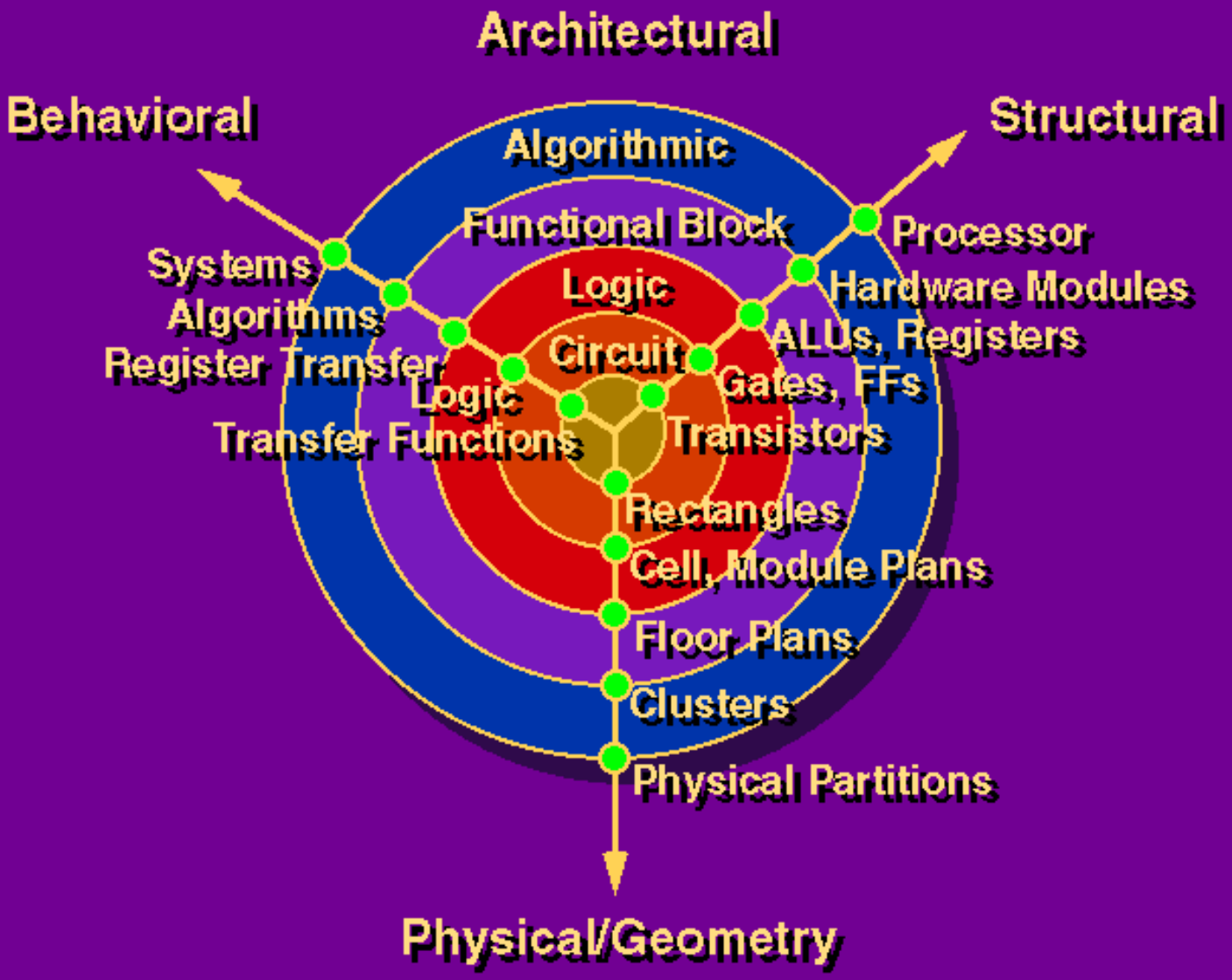
- VHDL is now undergoing international review to become an IEC standard

---

IEEE VHDL LRM

Gajski and Kuhn's Y Chart

VHDL HOME PAGE



[Walker85][Gajski83]

History of VHDL

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) is the product of a US Government request for a new means of describing digital hardware. The Very High Speed Integrated Circuit (VHSIC) Program was an initiative of the Defense Department to push the state of the art in VLSI technology, and VHDL was proposed as a versatile hardware description language.

The contract for the first VHDL implementation was awarded to the team of Intermetrics, IBM, and Texas Instruments in July 1983. However, development of the language was not a closed process and was subjected to public review throughout the process (accounting for Versions 1 through 7.1). The final version of the language, developed under government contract, was released as VHDL Version 7.2.

In March 1986, IEEE proposed a new standard VHDL to extend and modify the language to fix identified problems. In December 1987, VHDL became IEEE Standard 1076-1987. VHDL was again modified in September 1993 to further refine the language. These refinements both clarified and enhanced the language. The major changes included much improved file handling and a more consistent syntax and resulted in VHDL Standard 1076-1993.

IEEE VHDL LRM

Why Use VHDL?

VHDL HOME PAGE

- Allows for various design methodologies
- Provides technology independence
- Describes a wide variety of digital hardware
- Eases communication through standard language
- Allows for better design management
- Provides a flexible design language

Gajski and Kuhn's Y Chart

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

VHDL allows the designer to work at various levels of abstraction. Many of the levels are shown pictorially in the Gajski/Kuhn chart. Although VHDL does not support system description at the physical/geometry level of abstraction, many design tools can take behavioral or structural VHDL and generate chip layouts.

As an illustrative example, the next few slides will show a sample VHDL design process to demonstrate how a designer can move from an algorithmic behavioral description, to a register transfer (or data flow) description, to a gate level description. Although this chart is often referenced, this particular interpretation is found in [Walker85].

IEEE VHDL LRM

Sample VHDL Design Process

VHDL HOME PAGE

- **Problem: Design a single bit half adder with carry and enable**
- **Specifications**
  - **Passes results only on enable high**
  - **Passes zero on enable low**
  - **Result gets x plus y**
  - **Carry gets any carry of x plus y**

VHDL Models of Hardware

- VHDL models three very important and different facets of digital hardware
  - Behavior
  - Structure
  - Time
- VHDL makes no implicit assumptions about the hardware and takes a general approach
- VHDL combines all three facets of hardware description into a cohesive language

Why use VHDL?

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

VHDL is a powerful and versatile language and offers numerous advantages:

Design Methodology:

VHDL supports many different design methodologies (top-down, bottom-up, delay of detail) and is very flexible in its approach to describing hardware.

Technology Independence:

VHDL is independent of any specific technology or process. However, VHDL code can be written and then targeted at many different technologies.

Wide Range of Descriptions:

VHDL can model hardware at various levels of design abstraction. VHDL can describe hardware from the standpoint of a "black box" to the gate level. VHDL also allows for different abstraction-level descriptions of the same component and allows the designer to mix behavioral descriptions with gate level descriptions.

Standard Language:

The use of a standard language allows for easier documentation and the ability to run the same code in a variety of environments. Additionally, communication among designers and among design tools is enhanced by a standard language.

Design Management:

Use of VHDL constructs, such as packages and libraries, allows common elements to be shared among members of a design group.

Flexible Design:

VHDL can be used to model digital hardware as well as many other types of systems, including analog devices.

Sample VHDL Design Process

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Note that the code used in the example is "pseudo code" and not intended to follow VHDL syntax.

The requirement is to design a single bit adder with carry and enable functions. The inputs x, y, and enable are single bits with enable active high. The output of result is the bit addition of x and y and the carry output is the carry generated by the addition. When the enable line is low, the adder is to output zeroes.

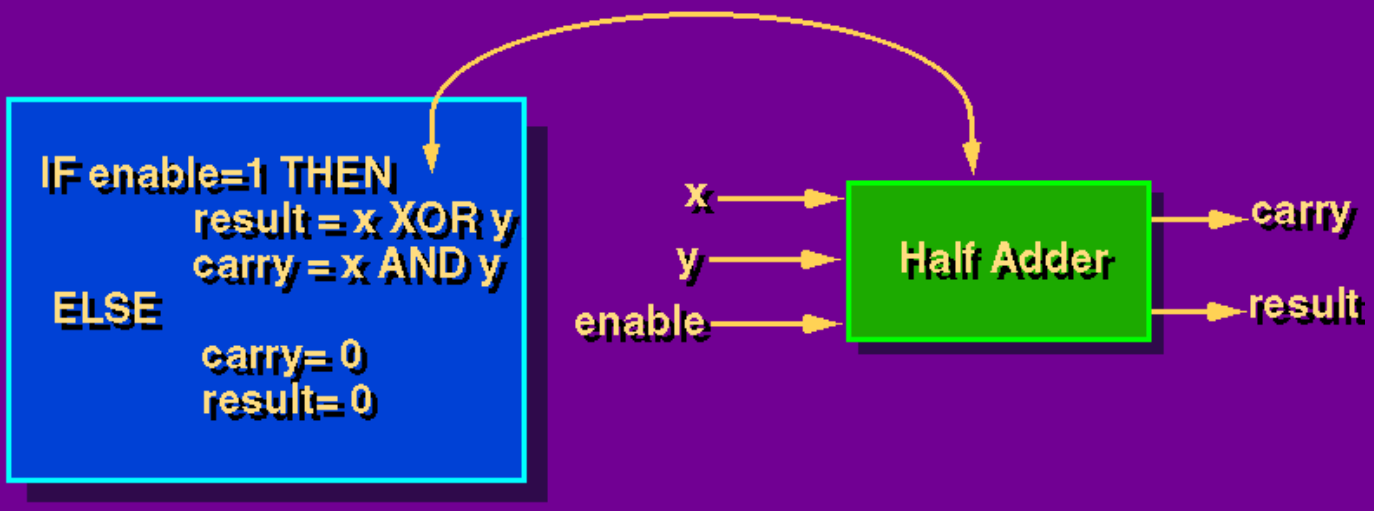
This sample design sequence is directly based on an example in [Navabi93].

IEEE VHDL LRM

Behavioral Specification

VHDL HOME PAGE

- **Starting with an algorithm, a high level description of the adder is created**



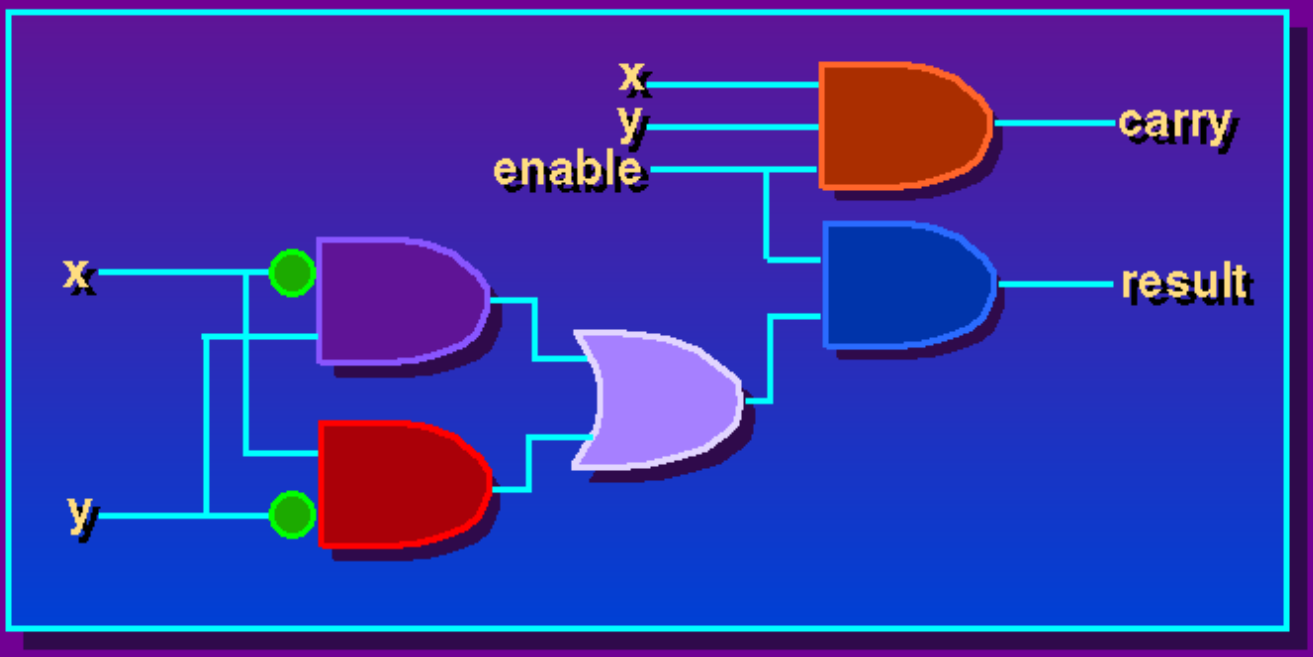
- **The model can now be simulated at this high level description to verify correct understanding of the problem**

IEEE VHDL LRM

Structural Specification

VHDL HOME PAGE

- Finally, a structural description is created at the gate level



- These gates can be pulled from a library of parts

VHDL Models of Hardware

-- Notes Page --

VHDL supports the three necessary facets for system modeling. Behavior is necessary to describe how system components respond to stimuli. Structure describes how the various subcomponents are connected to each other. The timing model provides a framework so that system events can occur in the correct order.

VHDL makes no assumptions about the underlying hardware. While the designer can add as many specifics as needed, VHDL will also operate with a general description only.

Behavioral Model

- Primary level of abstraction in VHDL is the entity
- In a behavioral description, the entity is defined by its responses to signals or input
- A behavioral model is similar to a "black box"
  - Interior is hidden from view
  - Behavior of the entity is defined by the relationship of the input to the output

VHDL BASICS

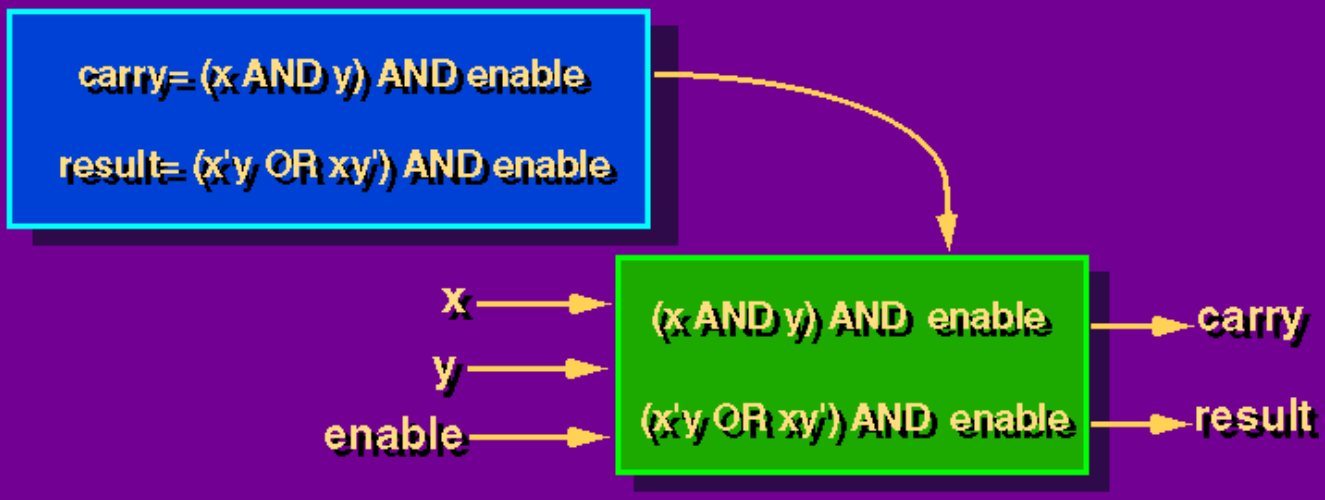
- Data Types
- Objects
- Sequential and concurrent statements
- Entity and architecture declarations
- Packages and libraries
- Attributes
- Predefined operators

IEEE VHDL LRM

Data Flow Specification

VHDL HOME PAGE

- With the high level description confirmed, logic equations describing the data flow are then created



- Again, the model can be simulated at this level to confirm the logic equations

Sample VHDL Design Process:

Structural Specification

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Finally, the logic equations may be mapped to specific logic gates. The models for these gates can come from many different libraries, and use specific or generic technologies. While this structural description ties together gate level logic, VHDL structural descriptions may be used to describe the interconnect of high level components as well (such as multiplexors, full adders, etc.).

Behavioral Model

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Unfortunately, the term *entity* has two meanings in the VHDL literature. First, it is used to signify the complete description for a component. Second, it used to refer to the VHDL construct in which a component's interface is described. The appropriate definition, however, is generally discernible from the context in which the term is used. In the context of this slide, for example, the student should use the first of the two definitions above.

No matter what level of abstraction is used for a VHDL model, the relationship between the model and its outside, as observed through its interface, must be described. A behavioral description of that relationship is generally the most abstract where specific details about a component's internal structure need not be made available, if in fact, they even exist at this point in the system's design.

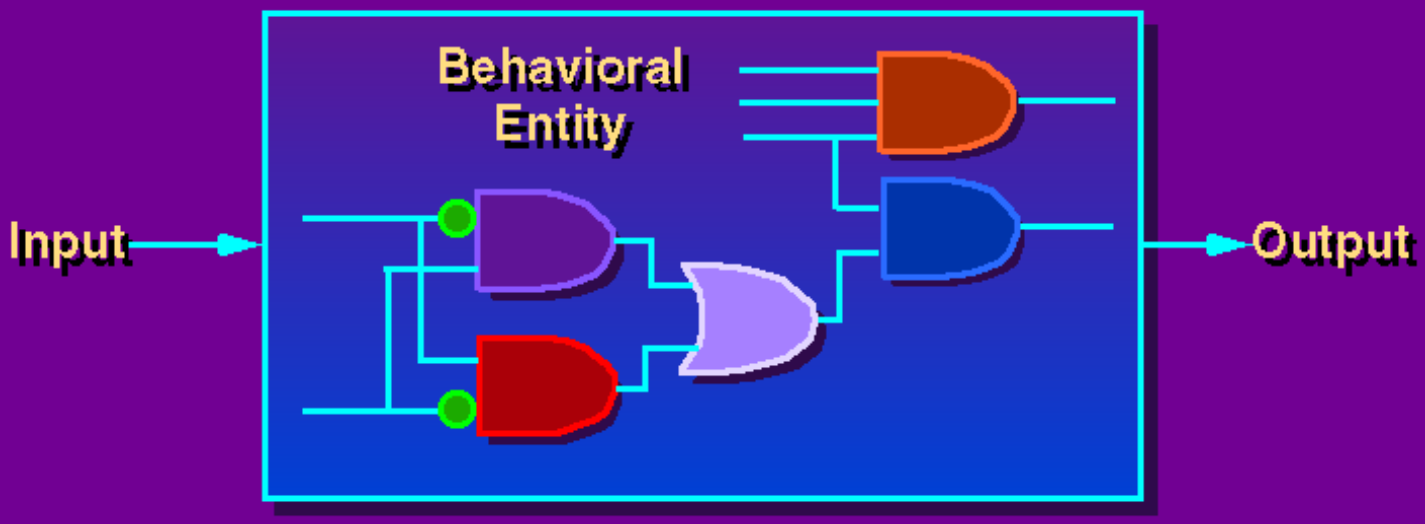
Each device in VHDL behaves as a *process* performing operations on the input to the device and writes to the output. This input and output information is carried in constructs called *signals* which are the chief means of communication between VHDL entities. Signals in VHDL are roughly similar to wires in the real world.

IEEE VHDL LRM

Structural Model

VHDL HOME PAGE

- VHDL can model the structure of digital devices
- Structural VHDL describes the arrangement and interconnection of components
- Structural descriptions support the use of predefined components
- Structural descriptions may connect simple gates or complex, abstract components

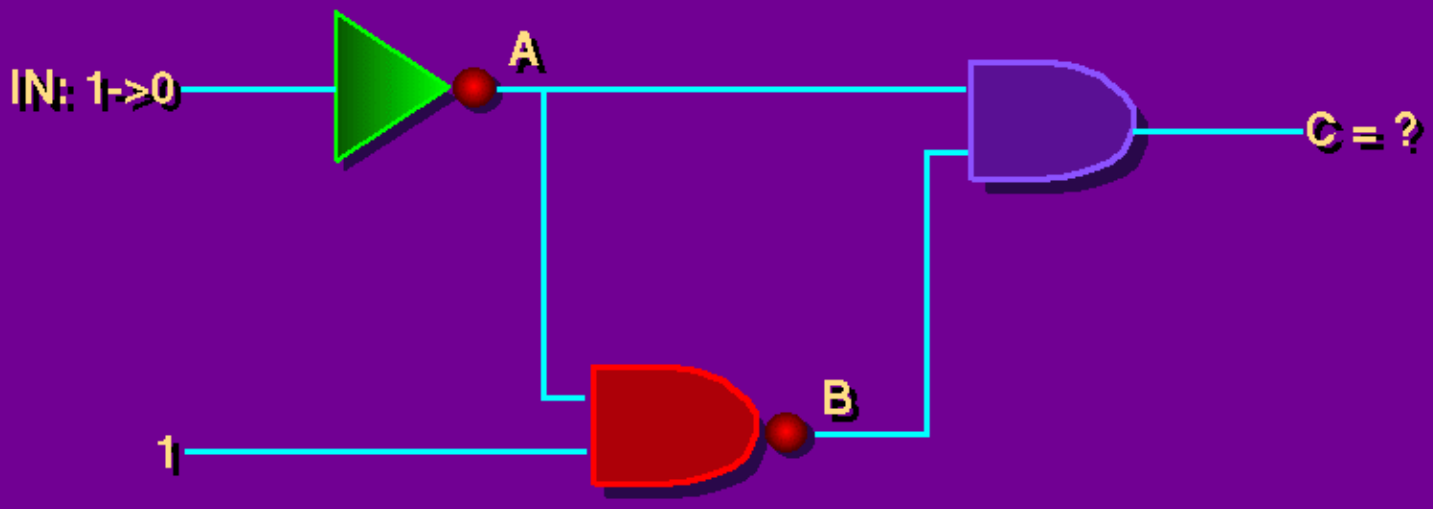


IEEE VHDL LRM

An Example With Delta Delay

VHDL HOME PAGE

- **What is the final output of C?**



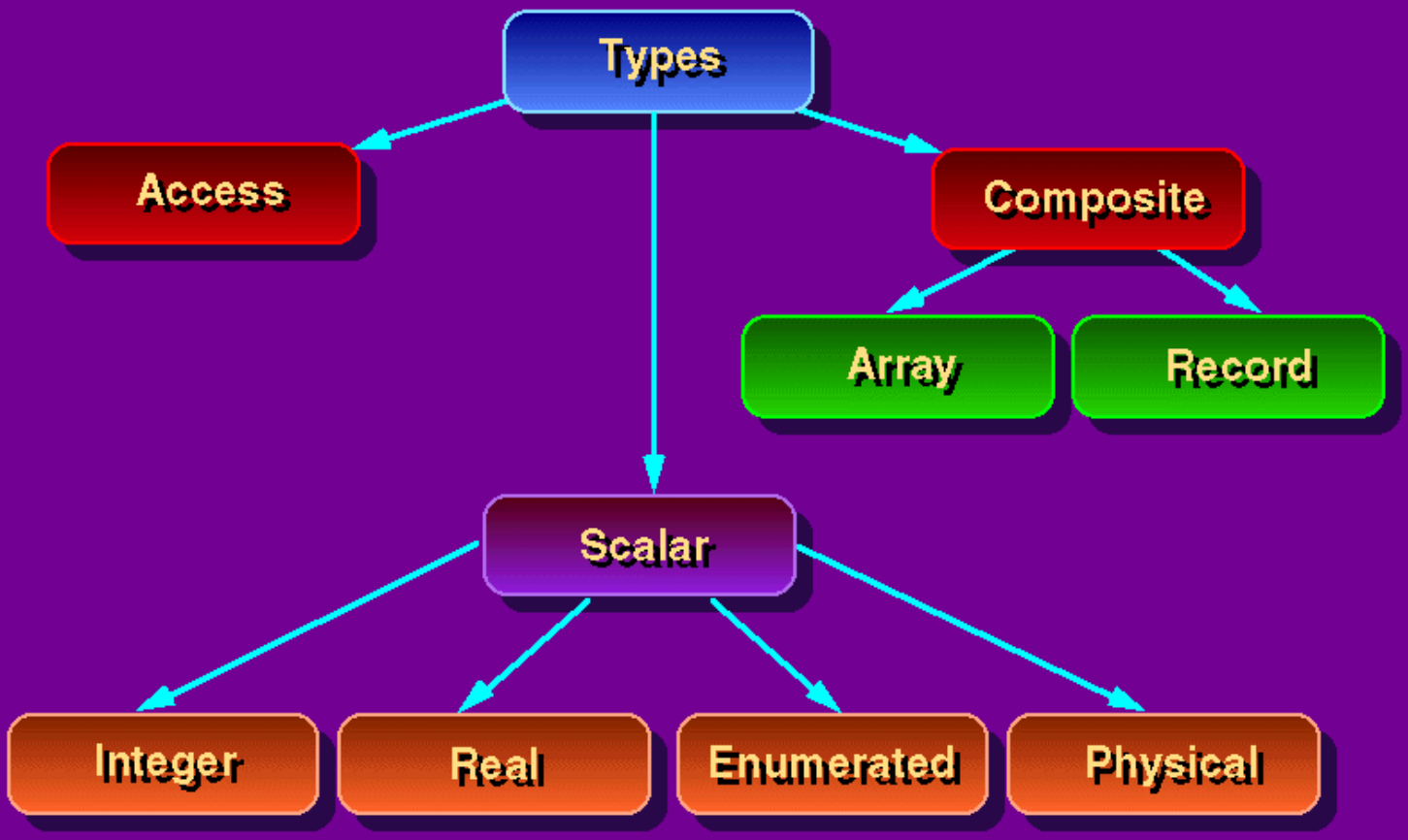
USING DELTA DELAY SCHEDULING

Time	Delta	Event
0 ns	1	IN: 1->0 eval inverter
	2	A: 0->1 eval NAND, AND
	3	B: 1->0 C: 0->1 eval AND
	4	C: 1->0
1 ns		

IEEE VHDL LRM

Data Types

VHDL HOME PAGE



Map Notes Help

IEEE VHDL LRM

Summary

VHDL HOME PAGE

- VHDL is a worldwide standard for the description and modeling of digital hardware
- VHDL gives the designer many different ways to describe hardware
- Familiar programming tools are available for complex and simple problems
- Sequential and concurrent modes of execution meet a large variety of design needs
- Packages and libraries support design management and component reuse

Delta Delay (cont.)

- The extra pulse on C may cause unexpected behavior in other components
- The unpredictable order of execution is not acceptable for modeling
- The delta delay establishes a clear order of events, with predictable and consistent behavior in execution

Delta Delay:

An example w/ Delta Delay

-- Notes Page --

This is the same example as before. However, each signal assignment will incur a one delta cycle delay.

The one to zero transition in *IN* occurs on the inverter just as before. *A* is then scheduled to be updated one delta cycle in the future. On the next delta cycle, *A* is updated, and now both the NAND and AND devices are evaluated concurrently and any resulting signal assignments will result in signals *B* and *C* being assigned new values one delta cycle in the future. When *B* and *C* change in the next delta cycle, however, the change in *B* triggers a second evaluation of the AND gate which results in an assignment of zero to *C*. *C* reaches this final value on the subsequent delta cycle.

[Perry94], pp. 22-24.

Data Types

-- Notes Page --

The three defined data types in VHDL are *access*, *scalar*, and *composite*. Note that VHDL 1076-1987 defined a fourth data type, *file*, but *files* were reclassified as *objects* in VHDL 1076-1993. In any case, *files* will not be discussed in this module but will be covered in the 'System Level VHDL' module included in this collection of educational modules.

Simply put, *access* types are akin to pointers in other programming languages, *scalar* types are atomic units of information, and *composite* types are arrays and/or records. These are explained in more detail in the next few slides. In addition, subtypes will also be introduced.

[Perry94], p. 74.

VHDL Data Types

Scalar Types

- **Integer**
  - Minimum range for any implementation as defined by standard: -2,147,483,647 to 2,147,483,647
  - Integer assignment example

```
ARCHITECTURE test_int OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a: INTEGER;
    BEGIN
      a := 1;   -- OK
      a := -1;   -- OK
      a := 1.0;   -- bad
    END PROCESS;
END TEST;
```

IEEE VHDL LRM

VHDL Objects

VHDL HOME PAGE

- There are four types of objects in VHDL
  - Files
  - Constants
  - Variables
  - Signals
- File declarations make a file available for use to a design
- Files can be opened for reading and writing
- Files provide a way for a VHDL design to communicate with the host environment

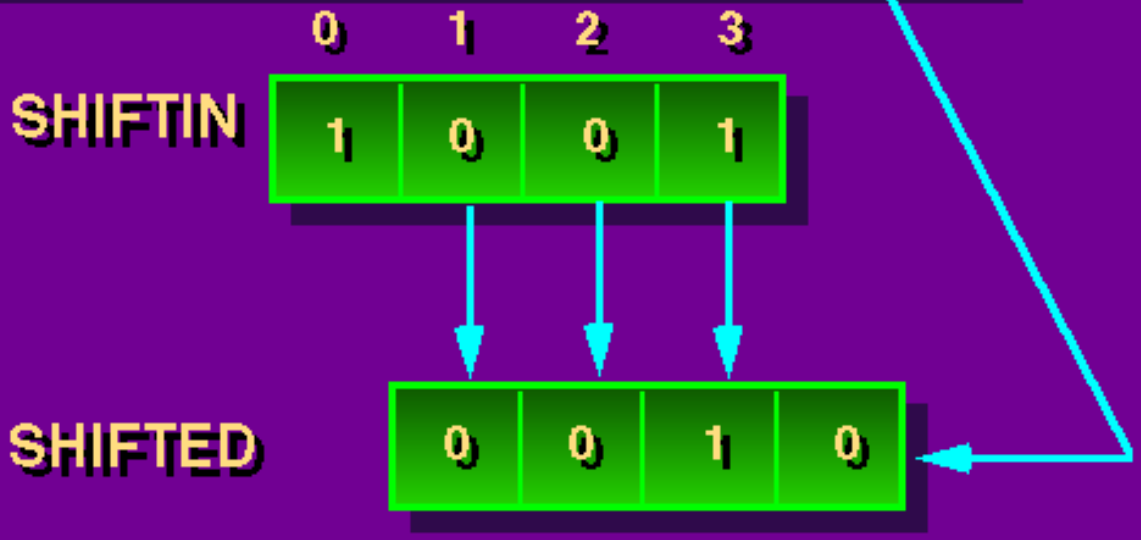
IEEE VHDL LRM

Some Explanations

VHDL HOME PAGE

The concatentation operator &

```
VARIABLE shifted, shiftin : BIT_VECTOR (0 TO 3) :
....
shifted := shiftin (1 TO 3) & '0';
```



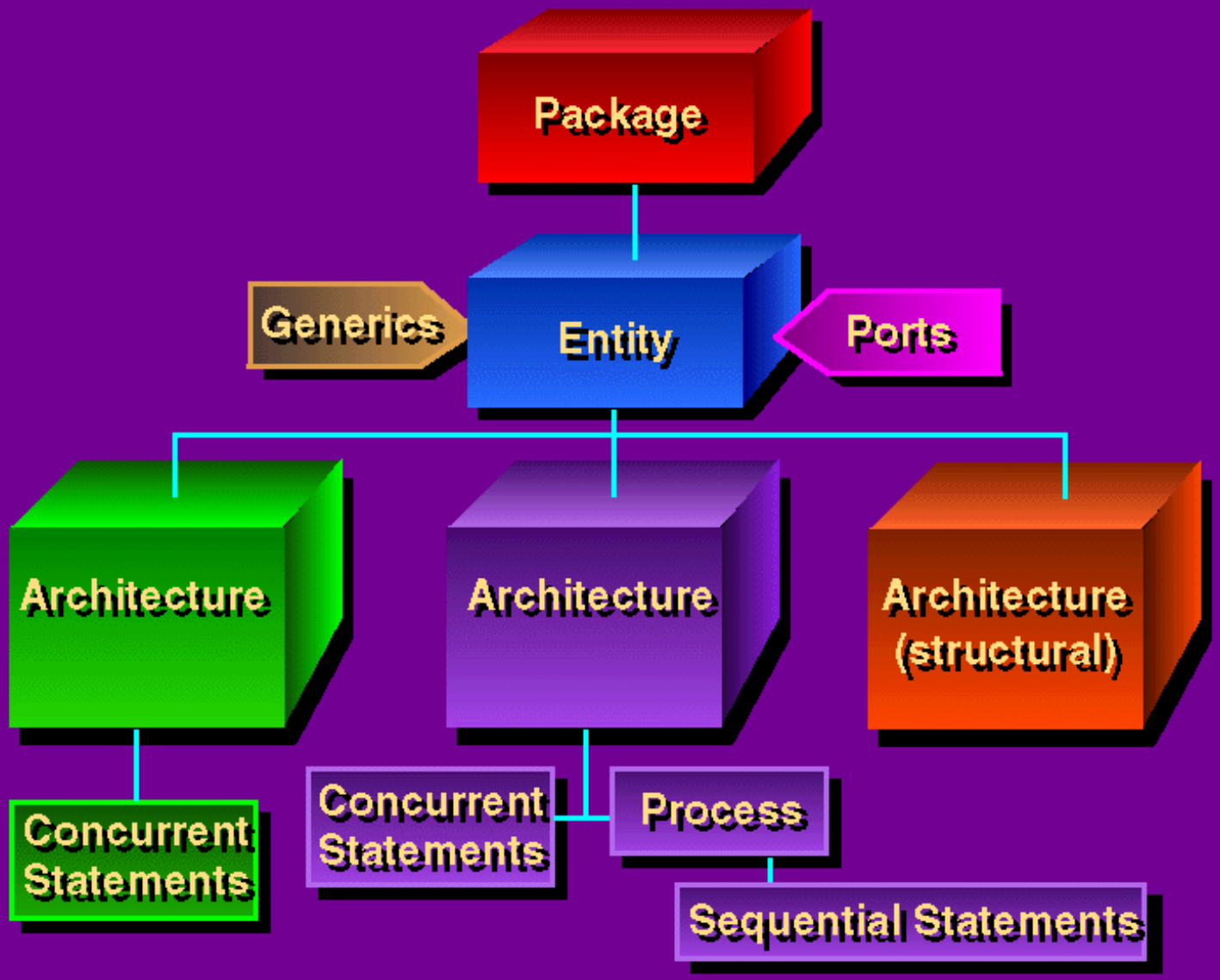
The exponentiation operator **

```
x := 5**5  — 5^5, OK
y := 0.5**3 — 0.5^3, OK
x := 4**0.5 — 4^0.5, bad
y := 0.5**(-2) — 0.5^(-2), OK
```

IEEE VHDL LRM

Putting It All Together

VHDL HOME PAGE



IEEE VHDL LRM

Attributes

VHDL HOME PAGE

- Language defined attributes return information about certain items in VHDL
  - Types, subtypes
  - Procedures, functions
  - Signals, variables, constants
  - Entities, architectures, configurations, packages
  - Components
- Attributes can be user-defined to handle custom situations (user-defined records, etc.)
- VHDL has several predefined attributes that are useful to the designer
- General form of attribute use

```
name ' attribute_identifier  ' read as "tick"
```

- Some example predefined attributes
  - X'EVENT -- evaluates TRUE when an event on signal X occurs
  - X'LAST_VALUE -- returns the last value of signal X
  - Y'HIGH -- returns the highest value in the range of Y
  - X'STABLE(t) -- evaluates TRUE when no event has occured on signal X in the past t" time


Map
Help
Notes

IEEE VHDL LRM

List of Operators

VHDL HOME PAGE

- **Logical operators**
  - **AND, OR, NAND, NOR, XOR, XNOR**
- **Relational operators**
  - =, /=, <, <=, >, >=
- **Addition operators**
  - **+, -, &**
- **Multiplication operators**
  - ***, /, mod, rem**
- **Miscellaneous operators**
  - ****, abs, not**

Predefined Operators:

Some Explanations

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This slide explains two of the less obvious operators.

The concatenation operator joins two vectors together. Both vectors must be of similar types. The example given above implements a logical shift left for this four-bit array by concatenating (or appending) a '0' to the vector resulting from a simple shift.

For the exponentiation operator **, the exponent must always be an integer. No real exponents are allowed, and negative exponents are allowed only with real numbers.

References

References:

[Walker85] Walker, Robert A. and Thomas, Donald E., "A Model of Design Representation and Syntheses", *22nd Design Automation Conference*, pp. 453-459, IEEE, 1985

[Gajski83] Gajski, Daniel D. and Kuhn, Robert H., "Guest Editors Introduction - New VLSI Tools", *IEEE Computer*, pp 11-14, IEEE, 1983

[Navabi93] Navabi, Z., VHDL: Analysis and Modeling of Digital Systems, McGraw-Hill, 1993.

[Perry94] Perry, D.L., VHDL, McGraw-Hill, 1994.

[USE/DA94] USE/DA Standards Survey, 1994.

[VI93] VHDL International Survey, 1993.

For further reading:

Bhasker, J., A VHDL Primer, Prentice Hall, 1995.

Calhoun, J.S., Reese, B.,, Class Notes for EE-4993/6993: Special Topics in Electrical Engineering (VHDL), Mississippi State University, http://www.erc.msstate.edu/mpl/vhdl-class/html, 1995

Coehlo, D.R., The VHDL Handbook, Kluwer Academic Publishers, 1989.

IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993.

Lipsett, R., C. Schaefer, C. Ussery, VHDL: Hardware Description and Design, Kluwer Academic Publishers, 1989.

RASSP Roadmap

-- Notes Page --

This diagram emphasizes the role of VHDL in the RASSP program. VHDL can be used for system definition, functional design, hardware-software partitioning, hardware design and hardware-software integration and test. The concept of virtual prototyping uses VHDL as the binding language of choice for all design paradigms.

The most common usage of VHDL prior to RASSP was in the area of hardware design. The RASSP program has extended VHDL's use to include executable requirements, performance modeling/system level design as well as system integration and test.

Sample VHDL Design Process:

Behavioral Specification

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

In the first stage of the design process, a high-level behavior of the adder is considered. This level uses abstract constructions (such as the *IF-THEN-ELSE* statement) to make the model more readable and comprehensible.

Simulation of the adder at this level proves correct understanding of the problem specifications of the adder. VHDL code for this adder will be shown later.

Sample VHDL Design Process:

Data-Flow Specification

-- Notes Page --

After the behavioral model is confirmed, a more specific model is formed. This model uses logic equations to describe the flow of data inside the model. Notice that the higher level construct of the *IF-THEN-ELSE* statement is gone.

Structural Model

-- Notes Page --

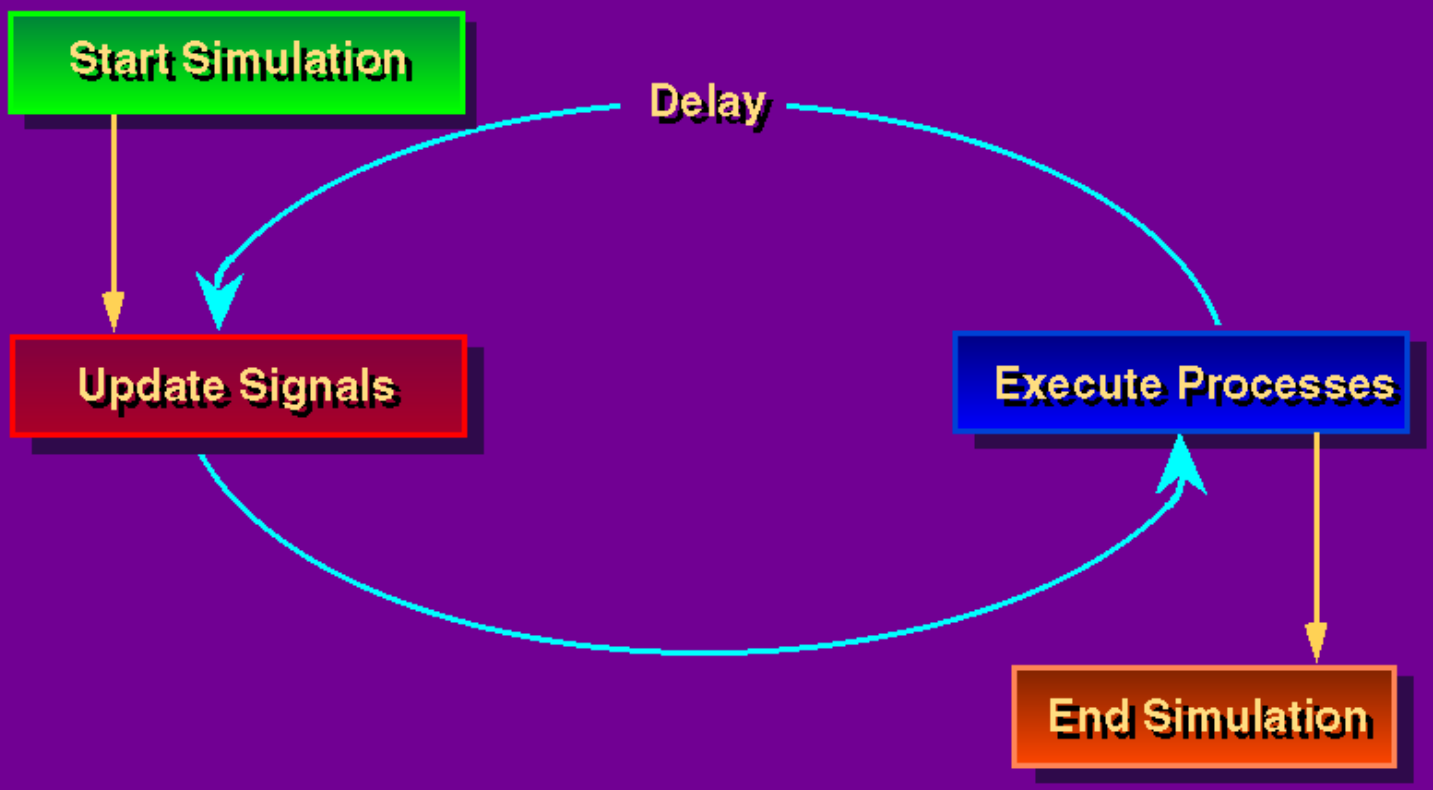
VHDL also supports descriptions based on a component's underlying internal structure. Structural VHDL allows for sub-components to be *instantiated* and interconnected. In fact, a structural description is similar to a netlist. Of course, the description of the sub-components can themselves be structural and/or behavioral in nature.

IEEE VHDL LRM

Timing Model

VHDL HOME PAGE

- **VHDL uses a simulation cycle to model the stimulus and response nature of digital hardware**



Timing Model

-- Notes Page --

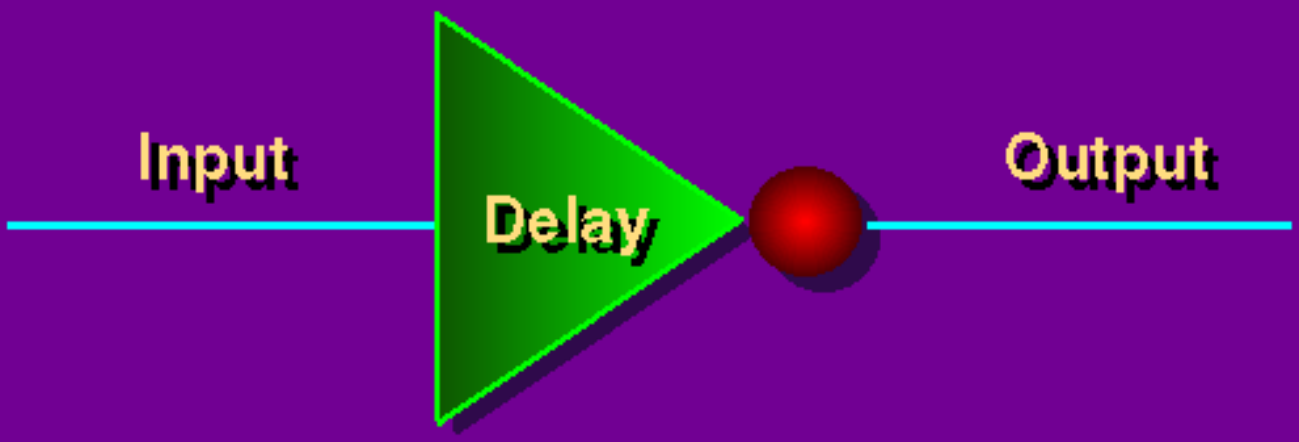
The VHDL timing model drives the stimulus and response sequence of digital hardware. At the start of a simulation, defined or implied initial values are assigned to all signals. All processes not suspended on *wait* conditions are executed concurrently until they reach their respective *wait* statements; these process executions will include signal assignment statements that assign new signal values after prescribed delays. After signals assume their new values, all processes examine their *wait* conditions to determine if they can proceed. Processes that can proceed will then execute concurrently until they all reach their respective *wait* conditions. This cycle continues until the simulation termination conditions are met or until all processes are suspended indefinitely because no new signal assignments are scheduled to unsuspend any process that is *waiting*.

IEEE VHDL LRM

Delay Types

VHDL HOME PAGE

- **Delay is created by scheduling a signal assignment for a future time**
- **Delay in a VHDL cycle can be of several types**
  - **Inertial**
  - **Transport**
  - **Delta**



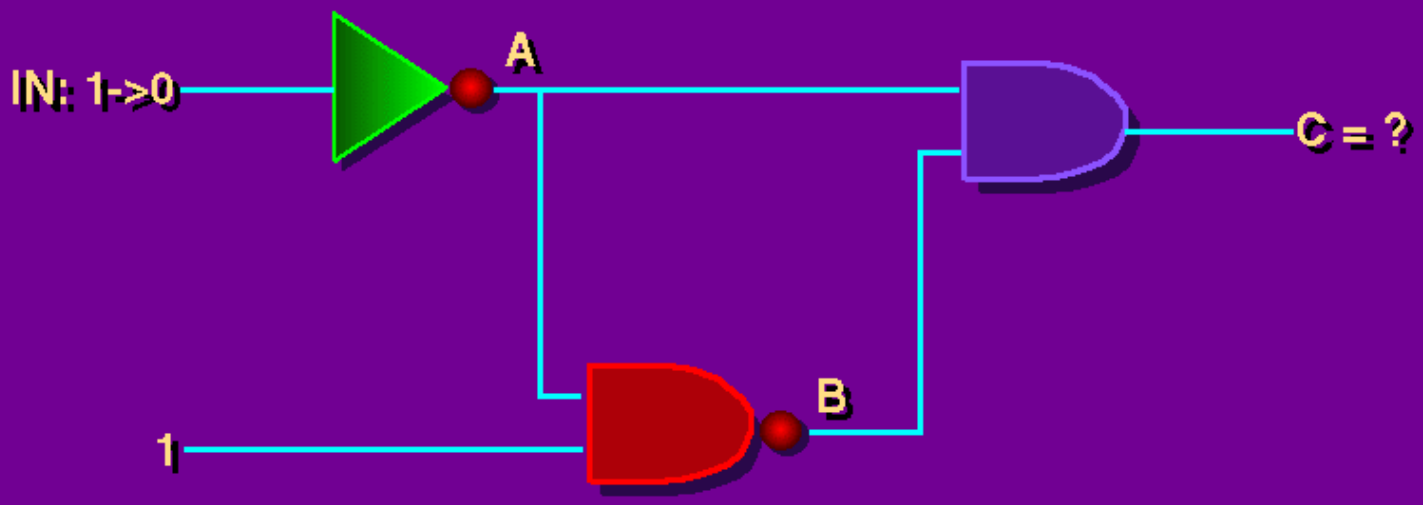
Delta Delay

IEEE VHDL LRM

An Example Without Delta Delay

VHDL HOME PAGE

- What is the output of C?



NAND gate evaluated first:
IN:1->0
A: 0->1
B: 1->0
C: 0->0

AND gate evaluated first:
IN:1->0
A: 0->1
C: 0->1
B: 1->0
C: 1->0

Delta Delay

-- Notes Page --

Without delta delay, the order of execution in this series of logic is uncertain. While the end result is the same, the extra pulse generated could cause other logic to trigger unexpectedly. Without a clear order of execution, VHDL would be a poor language for simulation.

The solution, in this case, is to make zero delay devices have an equal, but infinitesimal, delay, i.e. a *delta cycle* delay. The delta delay, as mentioned before, does not advance simulation time (i.e. no seconds, or ms, or ns, etc., are advanced). The delta delay is a scheduling device so that the simulator provides consistent and predictable behavior for models without specified delays.

VHDL Data Types:

Scalar Types

-- Notes Page --

Scalar objects can hold only one data value at a time. A simple example is the *integer* data type. Variables and signals of type *integer* can only be assigned integers within a simulator-specific (although the VHDL standard imposes a minimum) range.

In the above example, the first two variable assignments are valid since they assign integers to variables of type *integer*. The last variable assignment is illegal because it attempts to assign a real number value to a variable of type *integer*.

VHDL Data Types

IEEE VHDL LRM

Scalar Types 2

VHDL HOME PAGE

- **Real**
  - Minimum range for any implementation as defined by standard: -1.0E38 to 1.0E38
  - Real assignment example

```
ARCHITECTURE test_real OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a: REAL;
    BEGIN
      a := 1.3;  -- OK
      a := -7.5;  -- OK
      a := 1;  -- bad
      a := 1.7E13;  -- OK
      A := 5.3 ns;  -- bad
    END PROCESS;
END TEST;
```

VHDL Data Types

Summary

- VHDL has several different data types available to the designer
- Enumerated types are user defined
- Physical types represent physical quantities
- Arrays contain a number of elements of the same type or subtypes
- Records may contain a number of elements of different types or subtypes
- Access types are basically pointers
- Subtypes are user defined restrictions on the base type

VHDL Objects

-- Notes Page --

VHDL 1993 defines four types of objects, *files*, *constants*, *variables*, and *signals*.

The purposes of the *file* object are listed above. Files may be opened in read or write mode. Once a file is opened, its contents may only be accessed sequentially. A detailed description of the use of *file* objects is beyond this module and will be discussed further in the 'System Level VHDL module'.

IEEE VHDL LRM

VHDL Objects

Constants

VHDL HOME PAGE

- **Name assigned to a specific value of a type**
- **Allow for easy update and readability**

```
CONSTANT constant_name  :  type_name [:=value];

CONSTANT PI : REAL := 3.14;
CONSTANT SPEED : INTEGER := 100;
```

Sequential and Concurrent Statements

IEEE VHDL LRM

VHDL HOME PAGE

- VHDL provides two different types of execution: sequential and concurrent
- Different types of execution are useful for modeling of real hardware
  - Supports various levels of abstraction
- Sequential statements view hardware from a "programmer" approach
- Concurrent statements are order-independent and asynchronous

IEEE VHDL LRM

Delay Types

-- Notes Page --

VHDL HOME PAGE

There are several types of delay in VHDL, and understanding how delay works in a process is key to writing and understanding VHDL.

Simply put, any signal assignment in VHDL is actually a scheduling for a future value to be placed on that signal. When a signal assignment statement is executed, the signal maintains its original value until the time for the scheduled update to the new value. Any signal assignment statement will incur a delay of one of the three types listed in this slide.

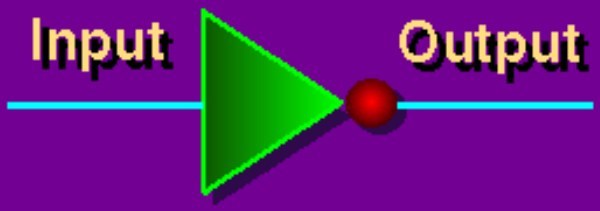
IEEE VHDL LRM

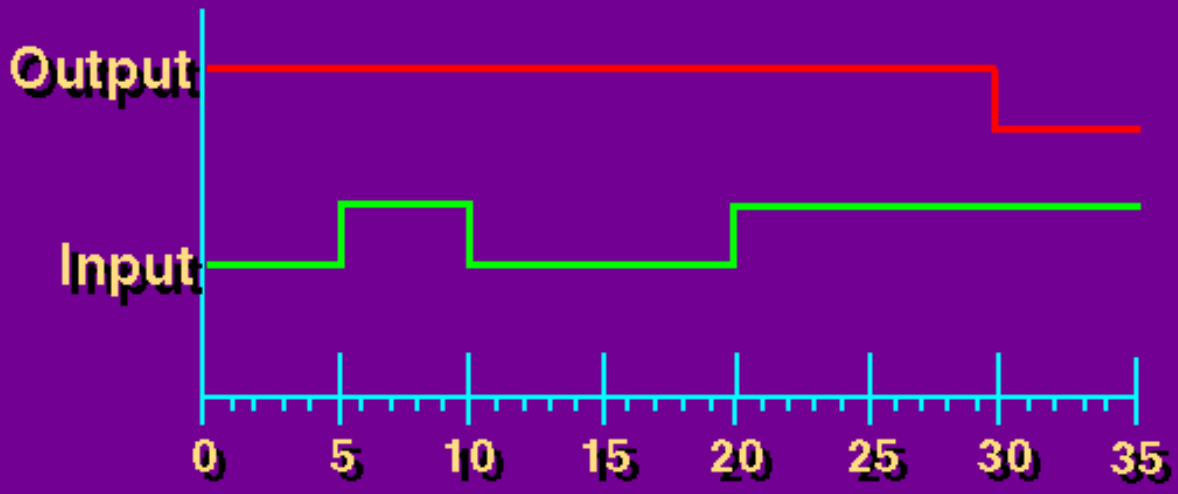
Inertial Delay

VHDL HOME PAGE

- Default delay type
- Allows for user specified delay
- Absorbs pulses of shorter duration than the specified delay

```
-- Inertial is the default
Output <= NOT Input AFTER 10 ns;
```





IEEE VHDL LRM

Delta Delay

VHDL HOME PAGE

- **Delta delay needed to provide support for concurrent operations with zero delay**
  - **The order of execution for components with zero delay is not clear**
- **Scheduling of zero delay devices requires the delta delay**
  - **A delta delay is necessary if no other delay is specified**
  - **A delta delay does not advance simulator time**
  - **One delta delay is an infinitesimal amount of time**
  - **The delta is a scheduling device to ensure repeatability**

Delta Delay:

An example w/o Delta Delay

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Let's assume that the above circuit does not specify any delays, and that there is no *delta delay* mechanism in the simulator behavior. In such a case, the order of execution of concurrent events will be arbitrary. In such a case, if the input to the inverter has a 1 to 0 transition, and the other input to the NAND gate is a constant 1, what is the output of C?

The final answer is, of course, the same. C eventually goes to 0. However, the transient behavior of the circuit depends on the order in which the gates are evaluated. If the NAND gate is evaluated first, no glitch is seen on the output of the AND gate. When A goes to 1, the output of the NAND gate, being evaluated first, goes to 0. When the AND gate is then evaluated, its output will also evaluate to 0, which is its final value.

However, if the AND gate is evaluated first, a glitch is generated because the NAND gate has not yet been updated to its new value. Therefore, C initially goes to 1 and will only go to 0 after the NAND gate drives its output to 0.

Therefore, if the order of execution is arbitrary, the behavior of the system may be unpredictable. This is generally not an acceptable situation for modeling.

[Perry94], pp. 22-24.

VHDL Data Types:

Scalar Types (cont.)

-- Notes Page --

A second simple example is the *real* data type. This type consists of the real numbers within a simulator-specific (but with a VHDL standard imposed minimum) range. The variable assignment lines marked OK are valid assignments. The first statement marked "bad" attempts to assign an integer to a real type variable, and the second "bad" statement is not allowed since the unit *ns* implies a physical data type.

VHDL Data Types

IEEE VHDL LRM

Scalar Types 3

VHDL HOME PAGE

- **Enumerated**
  - User defined range
  - Enumerated example

```
TYPE binary IS ( ON, OFF );
...some statements ...
ARCHITECTURE test_enum OF test IS
BEGIN
    PROCESS (X)
      VARIABLE a: binary;
    BEGIN
       a := ON; -- OK
       ... more statements ...
      a := off;  -- OK
      ... more statements ...
    END PROCESS;
END TEST;
```

VHDL Data Types

Subtypes

- **Subtype**
  - **Allows for user defined constraints on a data type**
  - **May include entire range of base type**
  - **Assignments that are out of the subtype range result in an error**
  - **Subtype example**

```
SUBTYPE name IS base type RANGE <user range>;
SUBTYPE first_ten IS INTEGER RANGE 0 TO 9;
```

VHDL Data Types:

Summary

-- Notes Page --

VHDL offers a variety of different data types to the VHDL modeler providing the flexibility to describe systems at various levels of information abstraction.

VHDL Objects:

Constants

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

VHDL *constants* are objects with permanently assigned values. The value of a *constant*, however, does not need to be assigned at the time the constant is declared; it can be assigned later in a package body if necessary.

The syntax of the *constant* declaration statement is shown above. The *constant* declaration includes the name of the *constant*, its *type*, and, optionally, its value.

Constant assignments can be deferred in the package declarations. The assignment can then be made in the package body.

IEEE VHDL LRM

VHDL Objects

Scoping Rules

VHDL HOME PAGE

- VHDL limits the visibility of the objects, depending on where they are declared
- The scope of the object is as follows
  - Objects declared in a package are global to all entities that use that package
  - Objects declared in an entity are global to all architectures that use that entity
  - Objects declared in an architecture are available to all statements in that architecture
  - Objects declared in a process are available to only that process
- Scoping rules apply to constants, variables, signals and files

VHDL Objects

Signals vs Variables (cont. 2)

- **This composite table of results clearly shows the differences between variables and signals**

		signals				variables	
Time	a	b	c	out_1	out_2	out_3	out_4
0	0	1	1	1	0	1	0
1	1	1	1	1	0	0	1
1+d	1	1	1	0	0		
1+2d	1	1	1	0	1		

- **The statements produce the same end result**

Sequential vs. Concurrent

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

In essence, VHDL is a concurrent language in that all *processes* execute concurrently. All VHDL execution can be seen as taking place inside *processes*; *concurrent signal assignment statements* have already been shown to be equivalent to one-line *processes*. Within a *process*, however, VHDL adheres to a sequential mode of execution where statements within a *process* are executed in "top-to-bottom' fashion until the *process* suspends at a *wait* statement.

This simultaneous support of concurrent and sequential modes allows great flexibility in modeling systems at multiple levels of design and description abstraction.

IEEE VHDL LRM

Sequential Statements

VHDL HOME PAGE

- Sequential statements run in top to bottom order
- Sequential execution most often found in behavioral descriptions
- Statements inside PROCESS execute sequentially

```
ARCHITECTURE sequential OF test_mux IS
BEGIN
    select_proc : PROCESS (x, y, selector)
    BEGIN
    IF (selector = '0') THEN
        z <= x;
    ELSIF (selector = '1') THEN
        z <= y;
    ELSE
        z <= "XXXX";
    END IF;
    END PROCESS select_proc;
END sequential;
```

IEEE VHDL LRM

Entity and Architecture Declarations

VHDL HOME PAGE

- An entity declaration describes the interface of the component
- PORT clause indicates input and output ports
- An entity can be thought of as a symbol for a component

```
ENTITY half_adder IS
     PORT (x, y, enable: IN bit;
           carry, result: OUT bit);

END half_adder;
```

IEEE VHDL LRM

Packages and Libraries

VHDL HOME PAGE

- User defined constructs declared inside architectures and entities are not visible to other entities
  - Subprograms, user defined data types, and constants can not be shared
- Packages and libraries provide the ability to reuse constructs in multiple entities and architectures

IEEE VHDL LRM

Libraries

VHDL HOME PAGE

- Increased complexity of devices requires configuration and revision control
- There is a need for using libraries of previous designs and modification of these libraries
- Design library is a set of files stored by the host operating system
- VHDL knows library only by logical name
  - Current design unit is compiled into the Work library
  - Both Work and STD libraries are always available

IEEE VHDL LRM

Attributes

-- Notes Page --

VHDL HOME PAGE

Attributes may be used to extract information about many different items in VHDL. Attributes can return various types of information. For example, an attribute can be used to determine the depth of an array, or its range, or its leftmost index, etc. Additionally, the user may define new attributes to cover specific situations. This capability allows user-defined constructs and data types to use attributes. Another example of the use of attributes is in assigning information to a VHDL construct, such as board location, revision number, etc.

A few examples of predefined VHDL attributes are shown above. Note that the apostrophe marking the use of an attribute is pronounced tick (i.e. 'EVENT is pronounced "tick EVENT").

Register Example

- The following example shows how attributes can be used to make an 8-bit register
- Specifications
  - Triggers on rising clock edge
  - Latches only on enable high
  - Has a data setup time of x_setup
  - Has propagation delay of pop_delay

```
ENTITY 8_bit_reg IS
    GENERIC (x_setup, prop_delay, : TIME);
     PORT (enable, clk : IN qsim_state;
             a : IN qsim_state_vector (7 DOWNTO 0);
             b : OUT qsim_state_vector (7 DOWNTO 0);
END 8_bit_reg;
```

- qsim_state type is being used - includes logic values 0, 1, X, and Z

IEEE VHDL LRM

Predefined Operators

VHDL HOME PAGE

- Operators manipulate the supplied data
- Most operators require both operands to be of the same type
  - Exception: operands of physical type may be multiplied and divided by integers and real numbers
  - Result of an expression with a relational operator will be Boolean

Predefined Operators:

List of Operators

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The above is the list of predefined operators in VHDL. The logical and relational operators are similar to those found in other languages. The addition operators are also familiar except for the concatenation operator which will be discussed in the next slide. The multiplication operators are also typical (e.g. the mod operator returns the modulus of the division and the rem operator returns the remainder). Finally, the miscellaneous operators provide some useful frequently used functions.

IEEE VHDL LRM

Inertial Delay

-- Notes Page --

VHDL HOME PAGE

Inertial delay is the default delay type if a signal assignment statement contains an *after* clause. This delay model assumes that a signal in a device has a certain amount of "inertia" that must be overcome before the signal can assume a new value. By default, this inertial delay is equal to the propagation delay time of the device. If the signal is of shorter duration than the inertial delay, then the pulse will not be seen at the output.

In the example above, the (inverted) value of the signal *Input* will be assigned to the signal *Output* AFTER 10 ns. Because inertial delay is the default, it does not need to be specified explicitly. In the example waveforms above, the 10ns inertial delay on the inverter will suppress the 5ns pulse on *Input* from being propagated to *Output*. That is, *Output* cannot have any pulses smaller than its inertial delay. Note that the *Input's* change at time 20ns is long enough to propagate to *Output*.

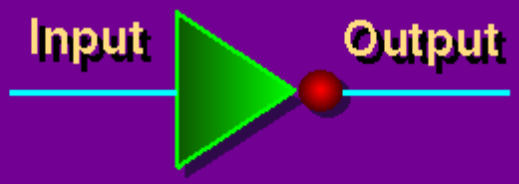
IEEE VHDL LRM

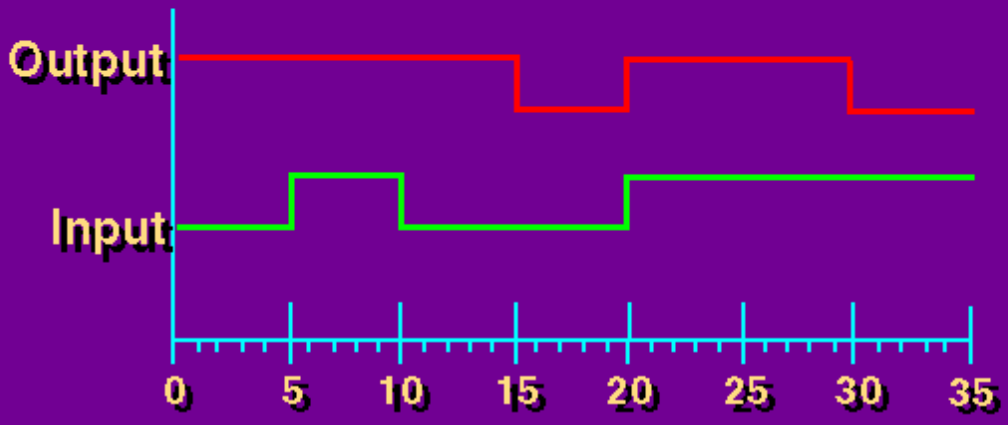
Transport Delay

VHDL HOME PAGE

- Must be explicitly specified by user
- Allows for user specified delay
- Passes all input transitions with delay

```
-- TRANSPORT must be specified
Output <= TRANSPORT NOT Input AFTER 10 ns;
```





IEEE VHDL LRM

Delta Delay

-- Notes Page --

VHDL HOME PAGE

VHDL encourages the designer to describe the hardware at whatever level is appropriate. Timing and delay information may not always be available when a system is first described in abstract forms. However, useful system simulations may be made at such levels so that models with no timing information must be supported.

VHDL maintains its sequential mode and concurrent mode semantics by the use of *delta* delay. A delta is essentially an infinitesimal, but quantized, unit of time. When a signal assignment is made with no explicit delay (i.e. no *after* clause), a delay of one *delta* cycle is assumed. This maintains the semantics that every signal assignment statement is actually the scheduling of a future value to be assigned to the signal, and the simulation cycle described earlier will function correctly. That is, all active processes continue to execute until they all suspend at *wait* statements at which point time advances (by either one delta cycle or by the minimum simulation time required for any signals to be able to assume new values).

The following examples will help to explain the concept of the delta delay further.

VHDL Data Types:

Scalar Types (cont.)

-- Notes Page --

The *enumerated* data type allows a user to specify the list of legal values that a variable or signal of the defined type may be assigned. As an example, this data type is useful for defining the various states of a FSM with descriptive names.

The designer first declares the members of the enumerated type. In the example above, the designer declares a new type *binary* with two legal values, ON and OFF.

Note that VHDL is not case sensitive. Typing reserved words in capitals and variables in lower case may enhance readability, however.

VHDL Data Types

IEEE VHDL LRM

Scalar Types 4

VHDL HOME PAGE

- **Physical**
  - Can be user defined range
  - Physical type example

```
TYPE resistence IS RANGE 0 to 1000000

UNITS
ohm;   -- ohm
Kohm = 1000 ohm;   -- 1 K
Mohm = 1000 kohm;   -- 1 M
END UNITS;
```

  - Time units are the only predefined physical type in VHDL

VHDL Data Types

Access Types

- **Access**
  - **Similar to pointers in other languages**
  - **Allows for dynamic allocation of storage**
  - **Useful for implementing queues, fifos, etc.**

VHDL Data Types:

Subtypes

-- Notes Page --

VHDL *subtypes* are used to constrain defined *types*. Constraints take the form of range constraints or index constraints. However, a *subtype* may include the entire range of the base *type*. Assignments made to objects that are out of the *subtype* range generate an error at run time. The syntax and an example of a *subtype* declaration are shown above.

VHDL Objects:

Scoping Rules

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Simple scoping rules determine where object declarations can be used. This allows the reuse of identifiers in separate entities within the same model without risk of inadvertent errors.

For example, a signal named *data* could be declared within the architecture body of one component and used to interconnect its underlying subcomponents. The identifier *data* may also be used again in a different architecture body contained within the same model.

VHDL Objects

Variables

- Variables are used for local storage of data
- Variables are generally not available to multiple components and processes
- All variable assignments take place immediately

```
VARIABLE variable_name : type_name [:=value];

VARIABLE opcode : BIT_VECTOR (3 DOWNTO 0) := "0000";
VARIABLE freq : INTEGER;
```

- Variables are more convenient than signals for the storage of data
- Variables may be made global

VHDL Objects

Signals vs Variables (cont. 1)

```
ARCHITECTURE variables OF test IS
    SIGNAL a: BIT:='0';
           b, c: BIT:='1';
BEGIN
    PROCESS (a, b, c)
    VARIABLE out_3, out_4 : BIT;
    BEGIN
    out_3 := a NAND b;
    out_4 := out_3 XOR c;
    END PROCESS;
END variables;
```

Time	a	b	c	out_3	out_4
0	0	1	1	1	0
1	1	1	1	0	1

VHDL Objects:

Signals vs Variables (cont. 2)

-- Notes Page --

In review, the final result for both of these implementations is the same. However, the example using *variables* reached its quiescent state immediately after the change in *a* rather than 2 delta cycles later, although the order of the two assignment statements became important. In addition, *variables* have less simulator overhead because there are no *waveforms* associated with them. In conclusion then, both *signals* and *variables* serve their purposes in VHDL well, but, although *signals* can often be used in place of *variables*, that practice leads to unnecessary simulation (and/or delta delays) and added simulator overhead.

Sequential Statements

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Statements in a VHDL *process* are executed sequentially. A *process* may also include a sensitivity list which is declared immediately after the PROCESS word. This sensitivity list makes the *process* execute when there is a transition on any of the specified *signals*. In the example above, the sensitivity list includes signals *x*, *y* and *selector*. The *process* can also be named; the process in the example above is named *select_proc*.

IEEE VHDL LRM

Concurrent Statements

VHDL HOME PAGE

- All concurrent statements occur simultaneously
- How are concurrent statements processed?
  - Simulator time does not advance until all concurrent statements are processed
  - End result is concurrent behavior with respect to simulator time
- Some concurrent statements
  - Block, process, assert, signal assignment, procedure call, component instantiation

IEEE VHDL LRM

Sequential Signal Assignments

VHDL HOME PAGE

- Signals can be both sequential and concurrent statements
- Signals require a delay even with sequential execution

```
ARCHITECTURE test1 OF test_mux IS
     SIGNAL a: BIT := '1';
     SIGNAL a: BIT := '0';
BEGIN
     PROCESS (...)
     BEGIN
       a <= b;
       b <= a;
     END PROCESS;
END test1;
```

- What are the final signal values for a and b in this case?
- Multiple assignments to the same signal in a PROCESS are allowed

```
ARCHITECTURE test1 OF test_mux IS
     SIGNAL a : BIT;
BEGIN
    PROCESS
     BEGIN
        a <= '1';
        a <= '0';
     END PROCESS;
END test1;
```

- **What is the value of a at the end of this process?**

Entity Declarations

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

In this slide the term *entity* refers to the VHDL construct in which a component's interface (which is visible to other components) is described. The first line in an *entity declaration* provides the name of the *entity*.

Next, the PORT statement indicates the actual interface of the entity. The *port* statement lists the *signals* in the component's interface, the direction of data flow for each *signal* listed, and *type* of each *signal*. In the above example, signals *x*, *y*, and *enable* are of direction IN (i.e. inputs to this component) and type *bit*, and *carry* and *result* are outputs also of type *bit*. Notice that if *signals* are of the same *mode* and *type*, they may be listed on the same line.

Particular attention should be paid to the syntax in that no semicolon is required before the closing parenthesis in the PORT declaration (or GENERIC declaration, for that matter, which is not shown here). The entity declaration statement is closed with the END keyword, and the name of the entity is optionally repeated.

IEEE VHDL LRM

Port Declaration

VHDL HOME PAGE

- PORT declaration establishes the interface of the object to the outside world
- Three parts of the PORT declaration
  - Name
  - Mode
  - Data type
- Sample PORT declaration:

```
ENTITY test IS
      PORT (name : mode data_type);
END test;
```

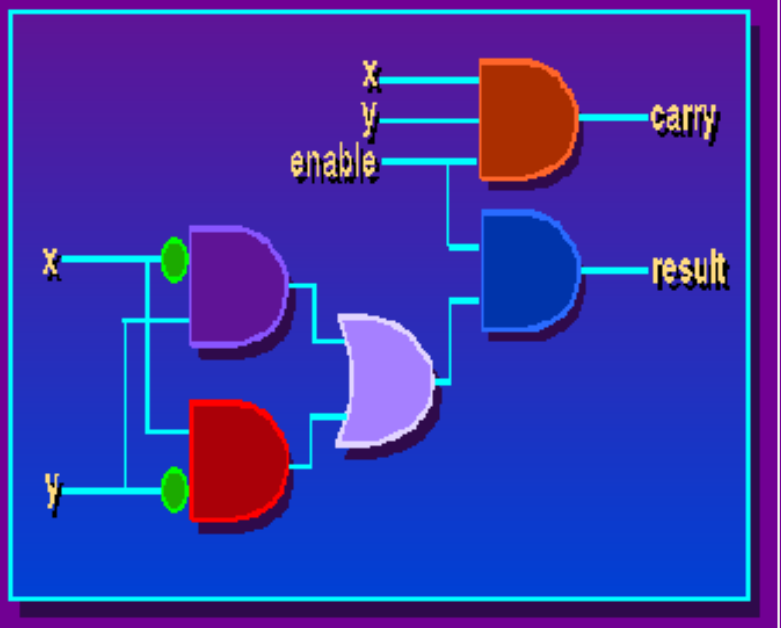
IEEE VHDL LRM

Architecture Declarations

VHDL HOME PAGE

- Architecture declarations describe the operation of the component
- Many architectures may exist for one entity, but only one may be active at a time
- An architecture is similar to a schematic of the component

```
ARCHITECTURE behavior1 OF
half_adder IS BEGIN
   PROCESS (enable, x, y)
   BEGIN
   IF (enable = '1') THEN
     result <= x XOR y;
     carry <= x AND y;
   ELSE
     carry <= '0'
     result <= '0';
   END PROCESS;
END behavior1;
```



Packages and Libraries

-- Notes Page --

VHDL provides the *package* mechanism so that user-defined *types, subprograms, constants, aliases,* etc. can be defined once and reused in the description of multiple VHDL components. VHDL *libraries* are collections of *packages, entities,* and *architectures*. The use of *libraries* allows the organization of the design task into any logical partition the user chooses (e.g. component libraries, package libraries to house reusable functions and type declarations).

IEEE VHDL LRM

Packages

VHDL HOME PAGE

- Items declared inside a package can be made visible to more than one entity or architecture
- Some valid statements for inclusion in the package declaration
  - Basic declarations
    - Types, subtypes
    - Constants
    - Subprograms
    - Use clause
  - Signal declarations
  - Attribute declarations
  - Component declarations

IEEE VHDL LRM

Use Clause

VHDL HOME PAGE

- Packages must be made visible before they can be used
- The USE clause makes packages visible to entities and architectures

```
 -- use on the binary_new and add_bits3 declarations
USE my_stuff.binary_new, my_stuff.add_bits3;

... ENTITY declaration...
... ARCHITECTURE declaration...
```

```
-- use all of the declarations in package my_stuff
USE my_stuff.ALL;

... ENTITY declaration...
... ARCHITECTURE declaration...
```

Libraries

-- Notes Page --

Increasingly complex VLSI technology requires configuration and revision control management. Additionally, it requires efficient reuse of components when applicable and revision of library components when necessary.

VHDL uses a library system to maintain designs for modification and shared use. VHDL refers to a library by an assigned logical name; the host operating system must translate this logical name into a real file name and locate it. The current design unit is always compiled into the *Work* library; the *Work* is implicitly available to the user with no need to declare it. Similarly, the predefined library STD does not need to be declared before its *packages* can be accessed via *use clauses*. The STD library contains the VHDL predefined language environment, including the package STANDARD which contains a set of basic data types and functions and the package TEXTIO which contains some text handling procedures.

Register Example (cont. 1)

- The following architecture is a first attempt at the register
- The use of 'STABLE detects for setup violations in the data input

```
ARCHITECTURE first_attempt OF 8_bit_reg IS
BEGIN
  PROCESS (clk)
  BEGIN
  IF (enable = '1') AND a'STABLE(x_setup) AND
       (clk = '1') THEN
            b <= a AFTER prop_delay;
  END IF;
  END PROCESS;
END first_attempt;
```

- What happens if clk or enable was 'X'?

IEEE VHDL LRM

Register Example (cont. 2)

VHDL HOME PAGE

- The following architecture is a second and more robust attempt
- The use of 'LAST_VALUE ensures the clock is rising from a 0 value

```
 ARCHITECTURE behavior OF 8_bit_reg IS
BEGIN
  PROCESS (clk)
  BEGIN
  IF (enable ='1') AND a'STABLE(x_setup) AND
             (clk = '1') AND (clk'LASTVALUE = '0') THEN
          b <= a AFTER prop_delay;
  END IF;
  END PROCESS;
END behavior;
```

- Elaboration of the IF-THEN-ELSE structure could define the behavior under all conditions

Predefined Operators

-- Notes Page --

There are several predefined operators in VHDL that perform various calculations on their operands. Because VHDL has strong data typing rules, most of the operators require both operands to be of the same type. A notable exception to this rule, however, is that physical types may be multiplied and divided by integers and real numbers with the result being a physical type. Similarly, although most operators generate results of the same type as one or both of their operands, the result of an expression with an relational operator is a Boolean value.

Transport Delay

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Because *transport* delay is not the default, it must be explicitly specified in the signal assignment statement as shown above. Note the use of the keyword TRANSPORT specifies that any spikes on the input will be passed to the output after the 10ns propagation delay.

In this example, the *Output* will be an inverted copy of *Input* delayed by the 10ns propagation delay regardless of the pulse widths seen on *Input*.

VHDL Data Types:

Scalar Types (cont.)

-- Notes Page --

The *physical* data type is used for values which have associated units. The designer first declares the name and range of the data type and then specifies the units of the type. Notice there is no semicolon separating the end of the TYPE statement and the UNITS statement. The line after the UNITS line states the base unit of of the type. The units after the base unit statement may be in terms of the base unit or another already defined unit.

The only predefined physical type in VHDL is *time*.

VHDL Data Types

IEEE VHDL LRM

Scalar Types 5

VHDL HOME PAGE

- **The predefined time units are as as follows**

```
   TYPE TIME IS RANGE

UNITS
   fs;  -- femtosecond
   ps = 1000 fs;  -- picosecond
   ns = 1000 ps;  -- nanosecond
   us = 1000 ns;  -- microsecond
   ms = 1000 us;  -- millisecond
   sec = 1000 ms;  -- second
   min = 60 sec;  -- minute
   hr = 60 min;  -- hour
   END UNITS;
```

VHDL Data Types

Composite Types 3

- **Records**
  - Used to collect one or more elements of a different types in single construct
  - Elements can be any VHDL data type
  - Elements are accessed through field name
  - Sample record statement

```
TYPE binary IS ( ON, OFF );
TYPE switch_info IS
   RECORD
     status : binary;
     IDnumber : integer;
END RECORD;

VARIABLE switch : switch_info;
switch.status := on;  -- status of the switch
switch.IDnumber := 30;  -- number of the switch
```

VHDL Data Types:

Access Type

-- Notes Page --

The VHDL *access* type will not be discussed in detail in this module; it will be covered more thoroughly in the 'System Level VHDL' module appearing in this collection of modules.

In brief, the *access* type is similar to a pointer in other programming languages in that it dynamically allocates and deallocates storage space to the object. This capability is useful for implementing abstract data structures (such as queues and first-in-first-out buffers) where the size of the structure may not be known at compile time.

VHDL Objects:

Variables

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This discussion about VHDL variables does not include *global* (aka *shared*) variables which were introduced in the 1076-1993 standard. The discussion of shared variables is deferred to the 'System Level VHDL Module'.

VHDL *variables* are used within *processes* for local storage of data. Because the scope of a VHDL *variable* is the process in which it is declared, it cannot be used to pass information to other VHDL processes or entities.

An important feature of the behavior of VHDL variables is that an assignment to a VHDL *variable* results in the variable assuming its new value immediately (i.e. no simulation time or delta cycles must transpire as is the case for VHDL *signals*). This feature allows the sequential execution of statements within VHDL *processes* where *variables* are used as placeholders for temporary data, loop counters, etc.

Examples of variable declarations and assignments are shown above. Note that when a variable is declared, it may optionally be given an initial value as well.

IEEE VHDL LRM

VHDL Objects

Signals

VHDL HOME PAGE

- Signals are used for communication between components
- Signals can be seen as real, physical signals
- Some delay must be incurred in a signal assignment

```
SIGNAL signal_name : type_name [:=value];
SIGNAL brdy : BIT;
SIGNAL output : INTEGER := 2;
```

IEEE VHDL LRM

VHDL Objects

Signals vs Variables

VHDL HOME PAGE

- **A key difference between variables and signals is the assignment delay**

```
ARCHITECTURE signals OF test IS
    SIGNAL a: BIT:='0';
           b, c: BIT:='1';
           out_1, out_2:  BIT;
BEGIN
    out_1 <= a NAND b;
    out_2 <= out_1 XOR c;
END signals;
```

Time	a	b	c	out_1	out_2
0	0	1	1	1	0
1	1	1	1	1	0
1+d	1	1	1	0	0
1+2d	1	1	1	0	1

VHDL Objects:

Signals vs Variables (cont. 1)

-- Notes Page --

In this example, *variables* are used to achieve the same functionality as the example in the previous slide. In this example, however, when there is a change in *a* at time 1, both *out_3* and *out_4* will also assume their new values at time 1 because they are *variables*, and VHDL *variable* assignment statements result in the new values being assumed immediately.

Also note, however, that in this example, the order in which the statements appear within the process is important because the two statements are executed sequentially, and the process will only be executed once as a result of the single change in *a*.

Concurrent Statements

-- Notes Page --

For a list of concurrent statements, there is no prescribed order of execution; the effect of simultaneous execution is achieved via the use of the VHDL timing model. Some of the *concurrent statement* types provided in VHDL are listed above.

Assignments

- **This example will show the concept of sequential and concurrent assignments**

```
ARCHITECTURE test1 OF
test_mux IS
    SIGNAL a : BIT := '1';
    SIGNAL b : BIT := '0';
BEGIN

...more statements...
   a <= b;
   b <= a;
...more statements...

END test1;
```

```
ARCHITECTURE test2 OF test_mux IS BEGIN
PROCESS (result)
   VARIABLE a : BIT := '1';
   VARIABLE b : BIT := '0';
BEGIN
...more statements...
   a := b;
   b := a;
...more statements...
   END PROCESS;
END test2;
```

- **What are the final signal values for a and b in both cases?**

Sequential Signal Assignments

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

VHDL *signal assignment statements* can appear as either *sequential* or *concurrent* statements. Outside of a *process*, they are concurrent statements; inside a process they are sequential statements. In either case, however, the assigned signal will assume its new value after some determined delay (of either some specified simulation time delay or one delta cycle)

The slide above provides two example. In the first example, assume that the sensitivity list for the process does not list either *a* or *b* (it could list some other signal, such as *swap_sig*). In that case, *a* and *b* will swap values one delta cycle after the process executes (essentially one iteration of the type of loop seen in the example two slides before, but the sensitivity list in this case prevents an endless cycle).

One important feature of *sequential signal assignment statements* is illustrated in the second example above. Note that each process only has one "driver" for each signal that have signal assignment statements within the process. It is, therefore, possible to have multiple assignment statements to the same signal within a process. Note that multiple assignment statements to the same signal would not be possible outside a process because they would then be concurrent signal assignment statements with separate "drivers", and a Bus Resolution Function" would be needed (Bus Resolution Functions will be discussed in the 'Behavioral VHDL' module).

In the second example above, then, the two assignments to *a* are executed sequentially. The first schedules a value of '1' to be assigned to *a* one delta cycle in the future. The second assignment statement then schedules a value of '0' also one delta cycle in the future and will override the assignment from the first statement because it was executed later. That is, the *waveform* for *a* will be modified as a result of executing the second *signal assignment statement* so that the assignment to take place one delta cycle in the future will have a value '0' instead of '1'.

Port Declaration

-- Notes Page --

The PORT declaration describes the interface of the *entity* to all other VHDL *entities*. There are three essential elements to the PORT declaration: the name, mode, and type of the signals in the interface. A fourth element (not shown above) in a port declaration is the optional initial value which may be assigned to each signal if there are no active drivers on the signal at the start of a simulation.

Note that signals declared in an entity's PORT declaration may sometimes be referred to as *ports*.

IEEE VHDL LRM

Name

VHDL HOME PAGE

- **The name of the port can be any valid VHDL identifier**
  - **Port signals of the same type or subtype may be on the same line and separated by commas**

```
PORT (input : direction data_type);

PORT (input, control, data : direction data_type);
```

Entity Declarations

- Generics may be added for readability, maintenance and configuration

```
ENTITY half_adder IS

    GENERIC (prop_delay : TIME := 10 ns);
    PORT (x, y, enable: IN bit;
          carry, result: OUT bit);

END half_adder;
```

- In this case, a generic called prop_delay was added to the entity and defined to be 10 ns

Architecture Declarations

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The architecture declaration describes the operation of the component. There can be multiple architectures described for each entity. However, for each *instantiation* of the entity, one of the possibly several architectures must be selected.

In the above example, the declaration starts with the keyword ARCHITECTURE followed by the name of the architecture (e.g. *behavior1*) and the name of the entity with which the architecture is associated. The keyword BEGIN marks the beginning of the architecture body which may include *concurrent signal assignment statements* and *processes*. Although not shown above, any *signals* that are used internally in the architecture description but are not found in the entity's *ports* are declared before the BEGIN statement of the architecture body. The keyword END marks the end of the architecture declaration.

Packages

-- Notes Page --

A *package* contains a collection of user-defined declarations and descriptions that a designer makes available to other VHDL entities. Items within a package are made available to other VHDL entities (including other packages) with a *use* clause. Some examples of possible package contents are shown above.

The next two slides will describe the two parts of a VHDL *package*, the *package declaration* and the *package body*.

IEEE VHDL LRM

Declaration

VHDL HOME PAGE

- An example of a package declaration

```
 PACKAGE my_stuff IS
    TYPE binary_new IS (ON, OFF);
    CONSTANT PI : REAL : = 3.14;
    PROCEDURE add_bits3 (SIGNAL a, b, en : IN BIT;
    SIGNAL temp_result, temp_carry : OUT BIT);
END my_stuff;
```

- Procedure body is defined in the "package body"

IEEE VHDL LRM

Package Body

VHDL HOME PAGE

- The package declaration contains only the declarations of the various items
- The package body contains subprogram bodies and other declarations not intended for use by other VHDL entities

```
PACKAGE BODY my_stuff IS
     PROCEDURE add_bits3 (SIGNAL a, b, en : IN BIT;
                SIGNAL temp_result, temp_carry : OUT BIT) IS
      BEGIN      -- this function can return a carry
            temp_result <= (a XOR b) AND en;
            temp_carry <= a AND b AND en;
      END add_bits3;

END my_stuff;
```

- How are the packages made visible to entities?

Packages:

Use Clause

-- Notes Page --

Packages are made visible to a VHDL description through the use of the *use clause*. This statement comes at the beginning of the entity or architecture file and makes the contents of a package available within that file.

The USE clause can select all or only part of a particular *package*. In the first example above, only the *binary* data type and *add_bits3* procedure are made visible. In the second example, the full contents of the *package* are made visible by use of the keyword ALL in the *use clause*.

Attributes:

Register Example (cont.)

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This first implementation of the 8-bit register uses the *'STABLE* attribute to determine if the input satisfies the setup time requirement of the register.

However, the example does not consider the possibility of *clk* assuming values other than '0' or '1' (i.e. 'X' and 'Z' are also valid states for a QSIM_STATE type signal).

Attributes:

Register Example (cont.)

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This second implementation adds a check for '0' to '1' transitions on clk by using the *'LASTVALUE* attribute on the signal *clk*.

IEEE VHDL LRM

Type of Data

VHDL HOME PAGE

- The type of data flowing through the port must be specified to complete the interface
- Data may be of many different types, depending on the package and library used
- Some data types defined in the Standard package
  - Bit, Bit_vector
  - Boolean
  - Integer, Real
  - Time

```
PORT (input : IN BIT_VECTOR (3 DOWNTO 0) );

PORT (output, brdy, data : OUT BIT);
```

Entity Declarations

-- Notes Page --

The entity statement can include additional information.

In this case, the GENERIC declaration creates a parameter to be passed to the architectures of this entity. The generic *prop_delay* is created with a default value of 10 ns. At component instantiation, however, *prop_delay* could be set to any value of type time. Any object declared in an entity's GENERIC declaration is available as a read-only object within any architecture of that entity. For example, an architecture for the entity above may contain a signal assignment of the form

a <= *b* after *prop*_delay;

Generics are discussed further in the 'Structural VHDL' module.

Packages:

Declaration

-- Notes Page --

This is an example of a *package declaration*. The declaration begins with the keyword PACKAGE and the name of the package followed by the keyword IS. VHDL declaration statements are then included, such as *type declarations*, *constant declarations*, and *subprogram declarations*. The *package declaration* lists the contents of the package. For many VHDL constructs, such as types, declarations are sufficient to fully define them. For a subprogram, however, the declaration only specifies the parameters required by the function or procedure; the operation of the subprogram appears later in the *package body*. The *package declaration* ends with END and the package name.

Packages:

Package Body

-- Notes Page --

The *package body* contains the functional descriptions for the subprograms declared in the corresponding *package declaration*.

Once a package is defined, its contents are made visible to VHDL entities and architectures via a *use clause* which is analogous to the *include statement* of some other programming languages.

VHDL Data Types:

Scalar Types (cont.)

-- Notes Page --

The predefined *time* units in the VHDL specified *Standard* package are shown here. The range of the time units may vary by simulator implementation but must at least include the defined integer range when measured in femtoseconds.

VHDL Data Types

Composite Types 1

- **Array**
  - Used to collect one or more elements of a similar type in a single construct
  - Elements can be any VHDL data type
  - Sample one-dimensional array (vector)

```
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;
```

0 ...element numbers... 31

0	...array values...	1

```
VARIABLE X:  data_bus;
VARIABLE Y:  BIT;

Y := X(12);  -- Y gets value of 13th element from the left
```

VHDL Data Types

Composite Types 2

- **Another sample one-dimensional array (using the DOWNTO order)**

```
TYPE register IS ARRAY (15 DOWNTO 0) OF BIT;
```

15 ...element numbers... 0

0	...array values...	1

```
VARIABLE X:  register;
VARIABLE Y:  BIT;

Y := X(4);  -- Y gets value of 5th element from the right
```

- **DOWNTO keyword orders elements from left to right, with decreasing element indices**

VHDL Data Types:

Composite Types (cont.)

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The second VHDL composite type is the *record*. An object of type *record* may contain elements of different types. Again, a *record* element may be of any data type, including another *record*.

A TYPE declaration is used to define a *record*. Note that the types of a *record's* elements must be defined before the *record* is defined. Also notice that there is no semi-colon after the word RECORD. The RECORD and END RECORD keywords bracket the field names. After the RECORD keyword, the *record's* field names are assigned and their data types are specified.

In the above example, a record type, *switch_info*, is declared. This example makes use of the *binary* enumerated type declared previously. Note that values are assigned to record elements by use of the field names.

VHDL Objects:

Signals

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Although *signal* assignments may resemble *variable* assignments syntactically, VHDL *signals* serve a different purpose. *Signals* are used to pass information directly between VHDL *processes* and *entities*. As has already been described, *signal* assignments require a delay before the *signal* assumes its new value. In fact, a particular *signal* may have a series of future values with their respective timestamps pending in the signal's *waveform*. The need to maintain a *waveform* results in a VHDL *signal* requiring more simulator resources than a VHDL *variable*.

VHDL Objects:

Signals vs Variables

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

To review, note that some delay must transpire after a VHDL signal assignment statement before the signal assumes its new value. Examples will be used in this and the next slide to illustrate the difference between signals and variables. The example shown above utilizes *signals*.

The table indicates the values for the various signals at the key times in the example. At time 1, a new value of 1 is observed on *a*. This causes the *out_1* assignment statement to be evaluated resulting in a 0 being assigned to *out_1*. At time 1+d (i.e. 1 plus 1 delta cycle), *out_1* assumes its new value causing the *out_2* assignment statement to be evaluated resulting in a 1 being assigned to *out_2*. At time 1+2d, *out_2* assumes its new value of 1. This example, then, requires 2 delta cycles to arrive at its quiescent state following a change to *a* (or *b*, for that matter)

Note that the two *signal assignment statements* above are actually *concurrent signal assignment statements* so that the order in which they appear in the model is not important. In each case, it is a change (or more accurately, a *transaction*) in one of the *signals* in the "right hand side" that results in a concurrent signal assignment statement being evaluated.

IEEE VHDL LRM

Assignments

-- Notes Page --

VHDL HOME PAGE

Several examples are presented here to illustrate the subtleties of sequential and concurrent execution. In the example on the left, the two signal assignment statements will execute concurrently. The resulting behavior of signals *a* and *b* will be the perpetually swapping of their values in delta time. This behavior results from each signal assignment causing a transition one delta cycle in the future. When each signal is then updated, the signal assignment statements will be evaluated again because each had a transition in its "right-hand-side". Each signal will then be assigned a new value seen in the subsequent delta cycle, and the cycle continues endlessly.

In the example on the right, the variable assignments execute sequentially because they are inside a *process* (the only place VHDL variables can actually exist). The final value of both *a* and *b* will be 0 since the first assignment will copy the contents of *b* into *a*, and the second assignment will not accomplish anything useful since *b* and *a* will have the same value by then.

Port Declaration:

Name

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The term "port" in the slide above actually refers to a *signal* in the port declaration.

IEEE VHDL LRM

Port Mode

VHDL HOME PAGE

- The port mode of the interface describes the direction of the data flow with respect to the component
- The five types of data flow are
  - In - data flows in this port and can only be read
  - Out - data flows out this port and can only be written to
  - Buffer - data flow can be in either direction but only one source is allowed at any one time
  - Inout - data flow can be in either direction with any number of sources allowed (implies a bus)
  - Linkage - data flow direction is unknown

IEEE VHDL LRM

Port Mode Examples

VHDL HOME PAGE

- Two examples of port mode use

```
PORT (input : IN data_type);
PORT (output, brdy, data : OUT data_type);
```

- In the first case, note that input can only be read by the component
- In the second case, the signals can only be written to by the component

Port Declaration:

Type of Data

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Finally, the port must indicate the *type* of data it will use. Any VHDL-defined standard type or user-defined type may be used in a port declaration. Note that a range specification may be declared if an unconstrained type is used in the type declaration. Some of the data *types* defined in the VHDL Standard package are listed above.

VHDL Data Types:

Composite Types

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

VHDL *composite* types consists of arrays and records. Each object of this data type can hold more than one value.

Arrays consist of many similar elements of any data type, including *arrays*. The *array* is declared in a TYPE statement. There are numerous items in an array declaration. The first item is the name of the array. Second, the range of the array is declared. The keywords TO and DOWNTO designate ascending or descending indices within the specified range, respectively. The third item in the array declaration is the specification of the data type in each element of the array.

In the example above, an array consisting of 32 *bits* is specified. Note that individual elements of the array are accessed by using the index number of the element as shown above. The index number corresponds to where in the specified range the index appears. For example, *X(12)* above refers to the thirteenth element from the left (since the leftmost index is 0) in the array.

VHDL Data Types:

Composite Types (cont.)

-- Notes Page --

This example illustrates the use the DOWNTO designator in the range specification of the array. DOWNTO specifies a descending order in array indices so that in the example above, *X(4)* refers to the fifth element from the right in the array (again, 0 is the farmost index but is on the right end in this example).

Port Declaration:

Port Mode

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The *mode* indicates the direction of the flow of data across that port. This flow of data is defined with respect to the component.

The five port modes available:

IN --

data flows in this port and the device can only read from this port

OUT --

data flows out this port and the device can write to this port. Note that until 1076-1993, a component could not read a signal it was itself driving through an OUT port.

BUFFER --

data can flow in either direction and is read/writable. However, only one source at a time can drive a buffer; this requires the ability to disconnect drivers and will be discussed in the 'Behavioral VHDL' module.

INOUT --

data can flow in either direction and is read/writable. Any number of sources are allowed to drive the inout port, but a Bus Resolution Function is then required to determine what values the signal will assume. Again, this issue will be covered in the 'Behavioral VHDL' module.

LINKAGE --

data flow direction is unknown. This mode indicates only that a connection exists.

Port Declaration:

Port Mode Examples

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

These two examples show the use of the port mode. The first case indicates a port of mode IN. This data is coming in the port and can only be read by the device. The second example shows a port of mode OUT where the signal may be "driven" by this component. Note that in the 1076-1987 VHDL Standard, a component could not read its own OUT ports; this was changed in the 1076-1993 VHDL Standard.

Structural VHDL - Module 2

IEEE VHDL LRM

VHDL HOME PAGE

Table of Contents

Structural VHDL - Module 2

This module was prepared as part of the RASSP Education & Facilitation effort.

Version 1.0

Structural VHDL

IEEE VHDL LRM

-- Notes Page --

VHDL HOME PAGE

Structural VHDL allows the designer to represent a system in terms of components and their interconnections. This module discusses the constructs available in VHDL to facilitate structural descriptions of designs.

Toolbar Functionality

	Takes the user up one hierarchical level in the presentation.
	Takes the user to the previous section of the presentation.
	Takes the user to the previous slide in the presentation.
	Takes the user to a listing of all slides with links to each slide.
	Takes the user to a visual representation of the organization of the slide presentation.
	Takes the user to a document, further explaining the information contained within the slide.
	Brings the user to this document, containing information on the use of the toolbar.
	Takes the user to the next slide in the presentation.
	Takes the user to the next section of the presentation.
	Takes the user down one hierarchical level in the presentation.

Structural VHDL - Module 2

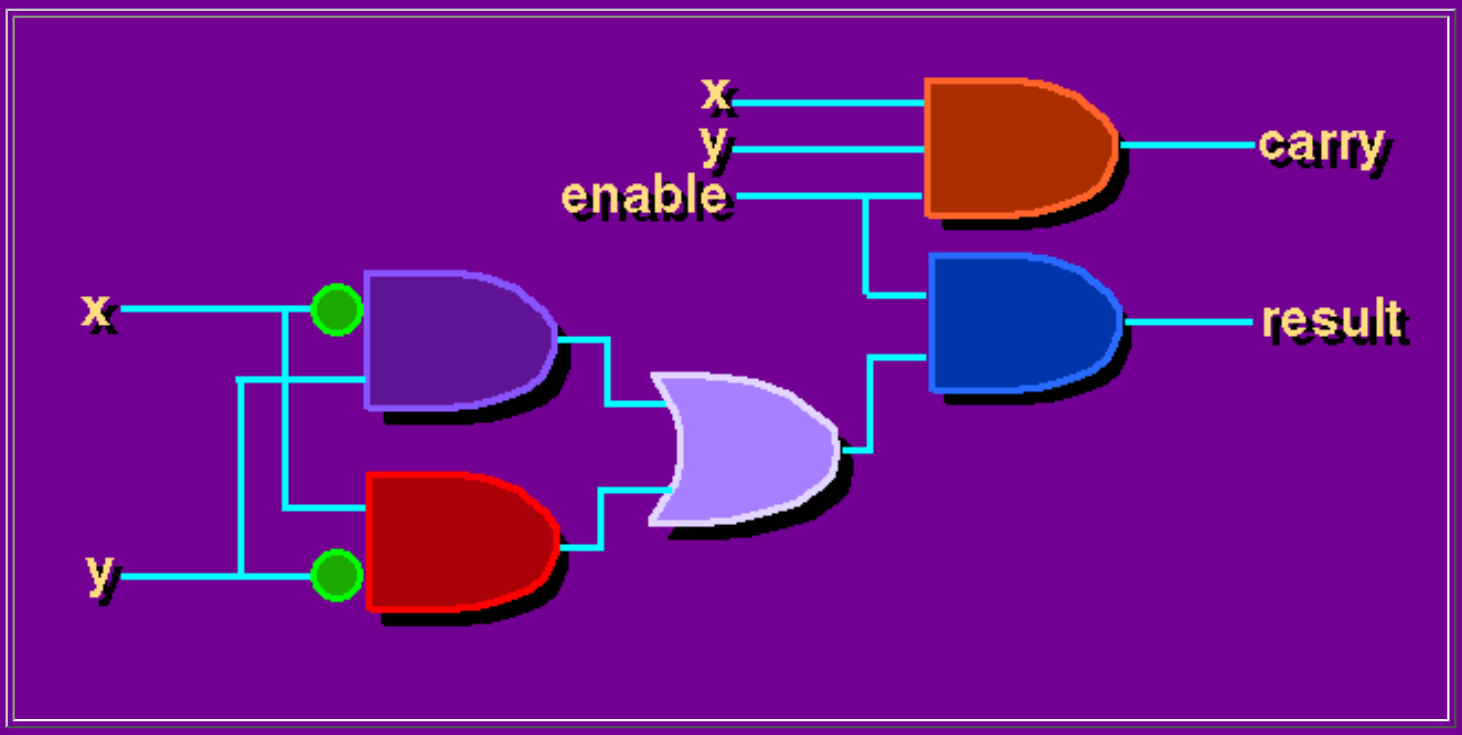
Outline

IEEE VHDL LRM

Introduction - Structured VHDL

VHDL HOME PAGE

- Circuits can be described like a netlist
- Components can be customized
- Large, regular circuits can be created



Structural VHDL - Module 2

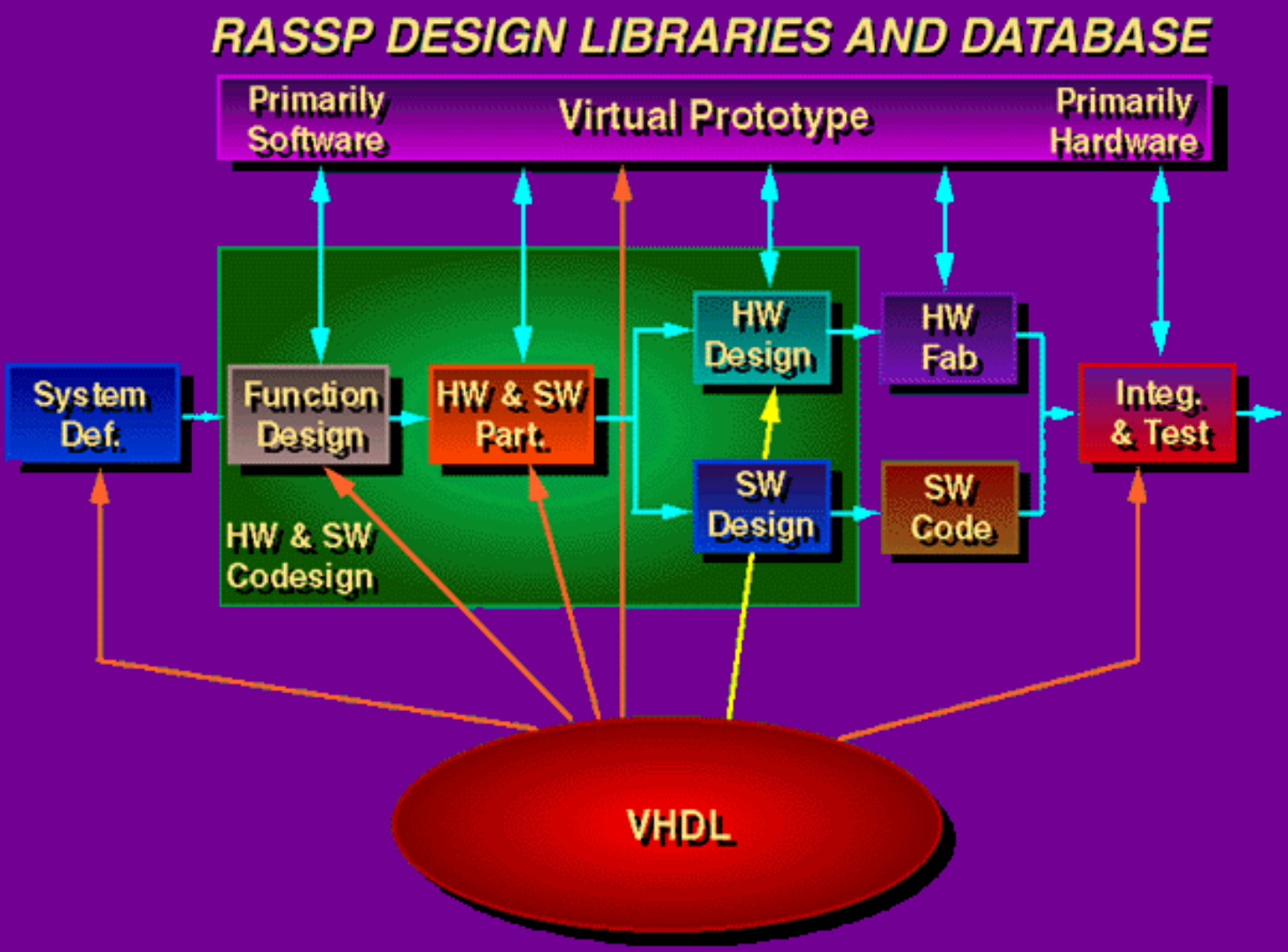
Module Goals

- Knowledge of structural VHDL concepts
- Understanding of structural VHDL constructs
- Comprehension of the uses for configuration

IEEE VHDL LRM

RASSP Roadmap

VHDL HOME PAGE



Map Notes Help

Module Goals

-- Notes Page --

The goals of the module are to introduce the concepts and constructs of structural modeling using VHDL. For example, VHDL has some powerful utilities that facilitate the design of systems with regular structures along with certain constructs to support configuration control. The goal of this module is to bring the student to the point where she/he will be able to write code using the concepts of structural design in VHDL.

Introduction

-- Notes Page --

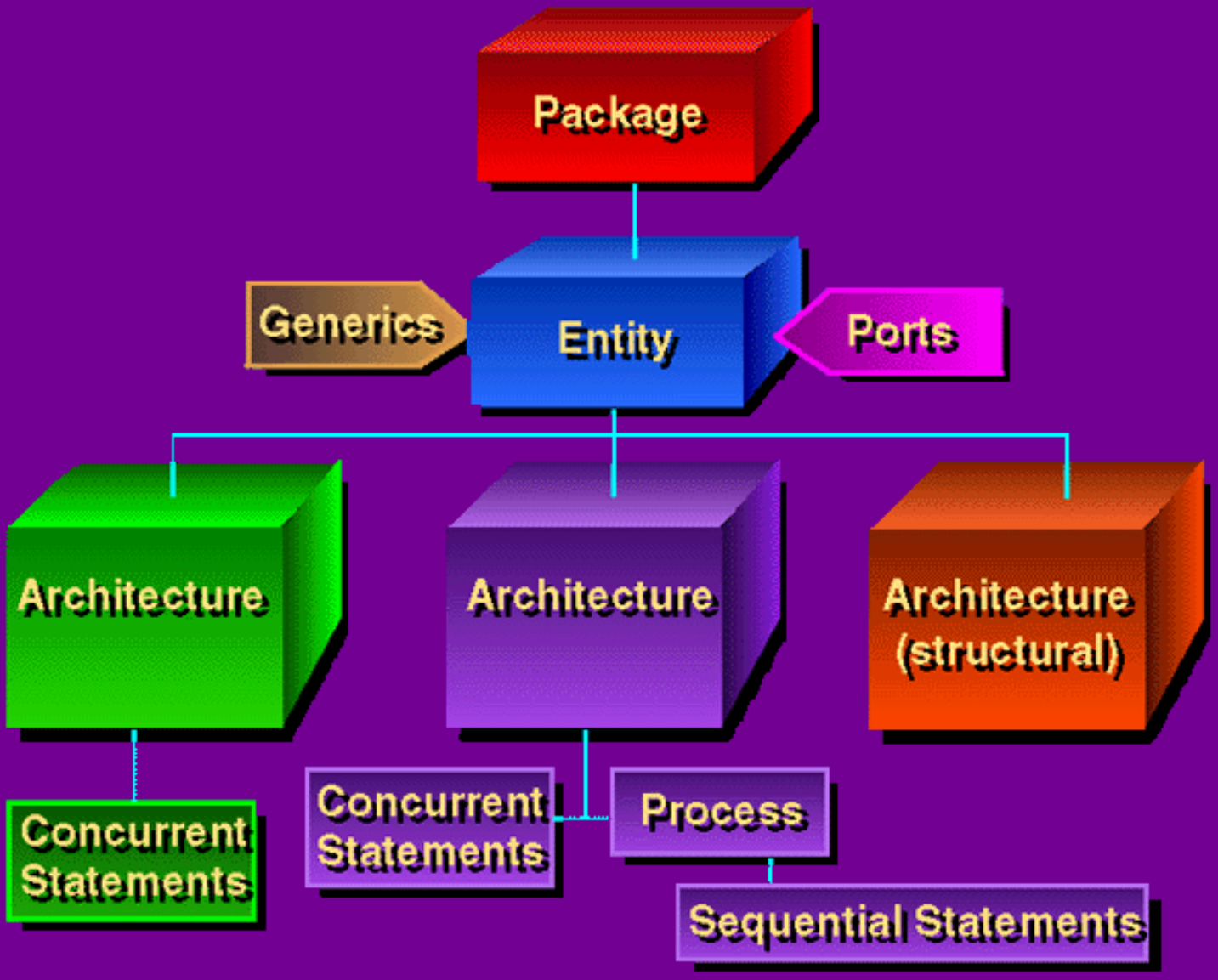
The role of structural VHDL is to describe circuits in terms of sub-components and interconnections. In this figure the simple logic elements are used to design a full adder. A structural description looks at the hardware as a netlist or schematic of the device; the components and interconnects are seen, but the internal function is hidden.

A structural description can tie together components of any complexity. A gate level description or the subunits of a microprocessor can be connected just as readily.

IEEE VHDL LRM

Putting It All Together

VHDL HOME PAGE



Map Notes Help

Putting It All Together

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This figure captures the main features of a complete VHDL model. A single component model is composed of one entity and one or many architectures. The entity represents the interface specification (I/O) of the component. It defines the components external view, sometimes referred to as its "pins".

The architecture(s) describe the function or composition of an entity. There are three general types of architectures. One type of architecture describes the structure of the design (right hand side) in terms of its sub-components and their interconnections. A key item of a structural VHDL architecture is the "configuration statement" which binds the entity of a sub-component to one of the possible several alternative architectures for that component.

A second type of architecture, containing only concurrent statements, is commonly referred to as a dataflow description (left hand side). Concurrent statements execute when data is available on their inputs. These statements can occur in any order within the architecture.

The third type of architecture is the behavioral description in which the functional and possibly timing characteristics are described using VHDL concurrent statements and processes. The process is a concurrent statement of an architecture. All statements contained within a process execute in a sequential order until it gets suspended by a wait statement.

Packages are used to provide a collection of common declarations,

constants, and/or subprograms to entities and architectures.

Generics provide a method to communicate static information to a architecture from the external environment. They are passed through the entity construct.

Ports provide the mechanism for a device to communication with its environment. A port declaration defines the names, types, directions, and possible default values for the signals in a component's interface.

Implicit in this figure is the testbench which is the top level of a self-contained simulatable model. The testbench is a special VHDL object for which the entity has no signals in its port declaration. Its architecture often contains construct from all three of the types described above. Structural VHDL concepts are used to connect the model's various components together, Dataflow and behavior concepts are often used to provide the simulation's start stop conditions, or other desired modeling directives.

Concepts of Structural VHDL

- Structural VHDL describes the arrangement and interconnection of components
  - Behavioral descriptions, on the other hand, define responses to signals
- Structural descriptions can show a more concrete relation between code and physical hardware
- Structural descriptions show interconnects at any level of abstraction

IEEE VHDL LRM

Component Instantiation

VHDL HOME PAGE

- Visibility of Components
- Instantiation statement
- Restrictions on instantiations

IEEE VHDL LRM

Visibility of Components

VHDL HOME PAGE

- Component instantiation is one of the building blocks of structural descriptions
- The component instantiation process requires component declarations and component instantiation statements
- Component instantiation declares the interface of the components used in the architecture
- At instantiation, *only* the interface is visible
  - The internals of the component are hidden

IEEE VHDL LRM

Generate Statements

VHDL HOME PAGE

- Uses of generate statements
- FOR-scheme
- IF-scheme

Visibility of Components

-- Notes Page --

A component represents an entity/architecture pair. A component instantiation statement defines a subcomponent of a design and associates signals with the ports of that subcomponent and associates values with generics of that subcomponent. An analogy to actual hardware would be the plugging of a hardware component into a board and making the electrical connections between the pins of the component and the circuit board.

In VHDL, the instantiation of components requires two mechanisms, the *Component Declaration* and the *Component Instantiation*. These will be shown in the subsequent slides.

IEEE VHDL LRM

Component Declaration

VHDL HOME PAGE

- Declares the interface of the component to the architecture

```
ARCHITECTURE test OF test_entity IS
    COMPONENT and_gate
        PORT (in1, in2: IN BIT;
              out1: OUT BIT);
     END COMPONENT;
...more statements...
```

- Necessary if the component interface is not declared elsewhere (package, library)

Component Declaration

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Before a component can be instantiated (i.e. "plugged in"), it must be declared either in a package, a library, or the architecture declaration region. An example of the syntax is shown on this slide for the component *and_gate* which requires two inputs, *in1* and *in2*, and one output, *out1*. Note that no information about how this gate works is given. Generics can also be included in the component declaration. The component declaration is used to select the component that will be used in subsequent component instantiations that appear in the architecture description.

IEEE VHDL LRM

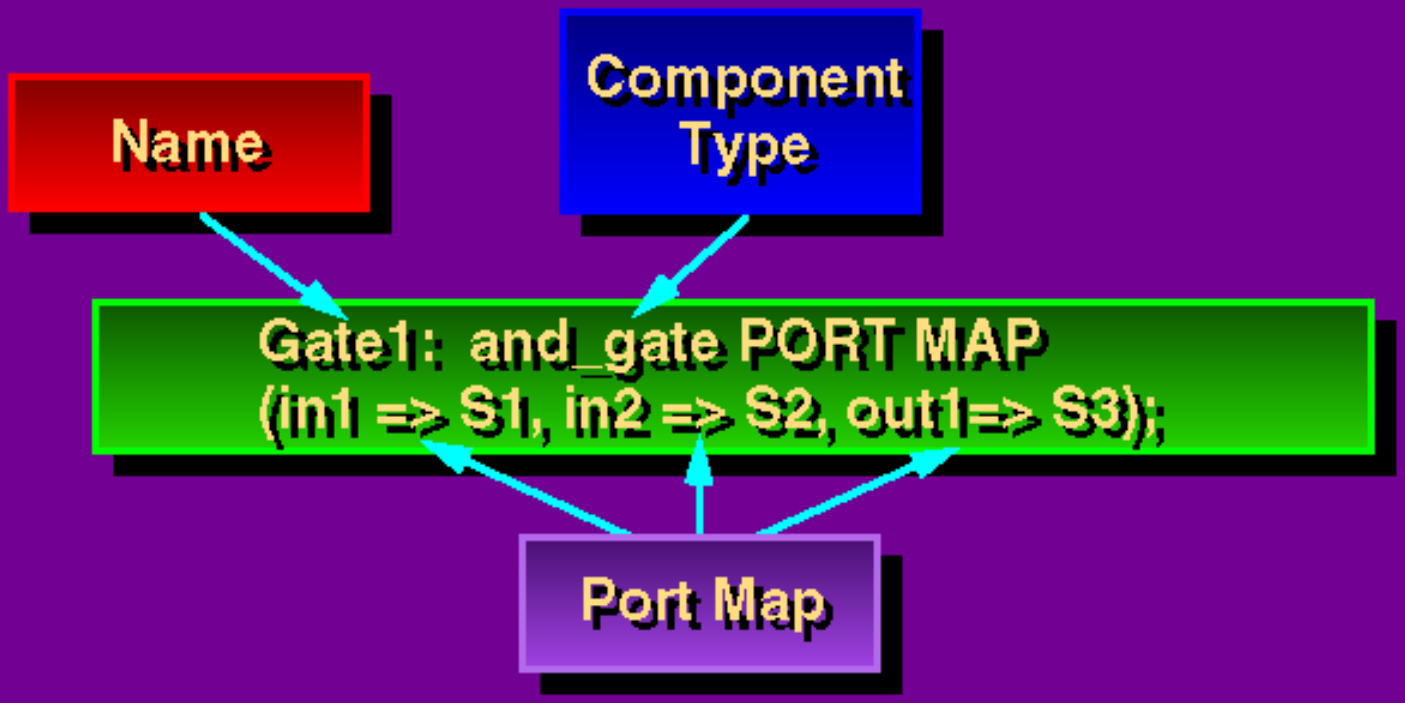
Instantiation Statements

VHDL HOME PAGE

- The instantiation statement maps the interface of the component to other objects in the architecture

```
ARCHITECTURE test OF test_entity IS
    COMPONENT and_gate
      PORT (in1, in2 : IN BIT;
                out1 : OUT BIT);
    END COMPONENT;
      SIGNAL S1, S2, S3 : BIT;
BEGIN
    Gate1 : and_gate PORT MAP (in1 => S1,
       in2 => S2, out1 => S3);
END test;
```

- The instantiation has 3 key parts
  - Name
  - Component type
  - Port map


Name
Component Type
Gate1: and_gate PORT MAP
(in1 => S1, in2 => S2, out1=> S3);
Port Map


S1
S2
in1
in2
out1
S3

Instantiation Statements

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

After the component is declared, it must be instantiated. The *and_gate* from the previous page is declared and instantiated in a design on this slide. The instantiation statement must specify at least three pieces of information to the architecture. First, the component must be named. Next, the type of the component instantiated must be specified. Finally, the mapping of the interface to other signals or ports is completed with the PORT MAP construct.

If the component contained generics, then a generic map could also be contained in the instantiation. The instantiation statement can be read as follows. The component named *Gate1* is of component type *and_gate* which is declared in the architecture declaration region of architecture *test*. The signals on its ports are tied as follows, input *in1* is tied to the circuit board signal *S1*, input *in2* is tied to *S2*, and the output *out1* is tied to circuit board signal *S3*.

In this example, each signal is explicitly associated to a port on the component. The use of the arrows makes the association completely clear. VHDL also allows for positional association which will be shown in subsequent slide.

IEEE VHDL LRM

Components from Packages

VHDL HOME PAGE

- Component declarations may be made inside packages
  - Components do not have to be declared in the architecture body

```
PACKAGE my_stuff IS
   COMPONENT and_gate
      PORT (in1, in2 : IN BIT;
            out1 : OUT BIT);
   END COMPONENT;
END PACKAGE my_stuff;
```

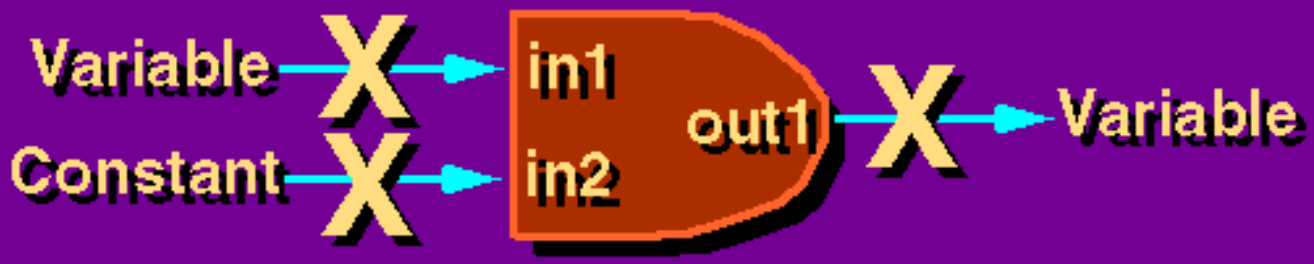
```
USE Work.my_stuff.ALL;
ARCHITECTURE test OF test_entity IS
     SIGNAL S1, S2, S3 : BIT;
BEGIN
   Gate1 : and_gate
     PORT MAP (S1, S2, S3);
END test;
```

IEEE VHDL LRM

Restrictions on Instantiation

VHDL HOME PAGE

- An actual can *only* be a signal or a formal port declared in the entity
  - A port on a component is known as a *local* and must be matched with a compatible *actual*
- The interface of the instantiated component must match the connecting objects



IEEE VHDL LRM

Uses of Generate Statement

VHDL HOME PAGE

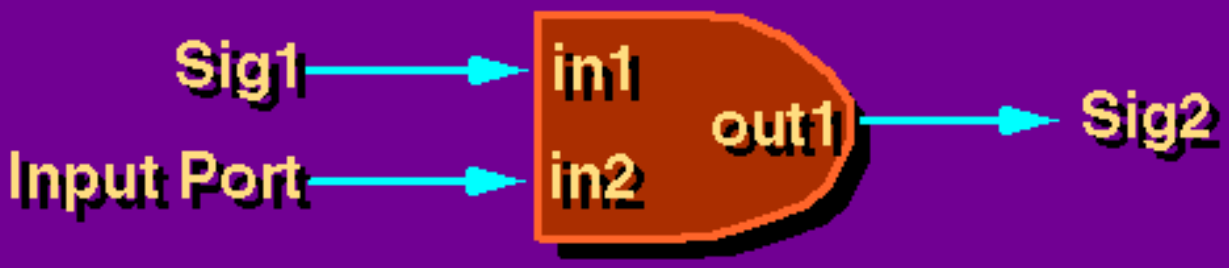
- Some structures in digital hardware are repetitive in nature (RAM, ROM, registers)
- VHDL provides the GENERATE statement to automatically create regular hardware
- Any VHDL concurrent statement may be included in a GENERATE statement, including another GENERATE statement

IEEE VHDL LRM

Rules for Actuals and Locals

VHDL HOME PAGE

- VHDL has two main restrictions on the association of locals with actuals
  - Local and actual must be of same data type
  - Local and actual must both be readable and writable
    - Locally declared signals are both and can connect to any local port



IEEE VHDL LRM

Configuration and Binding

VHDL HOME PAGE

- Need for configuration
- Configuration statement

Uses of Generate Statement

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Structural descriptions can more clearly indicate the nature of the physical hardware used to create the component. However, several digital devices have large regular structures that can be tedious to implement. Using a behavioral description or a long structural description could mask the regular structure of these devices.

VHDL provides the GENERATE statement to automatically create such structures. The GENERATE statement can be used in conjunction with any VHDL concurrent statement to create many repetitive objects. A GENERATE statement may even include other GENERATE statements for more complex devices. Some common examples include the instantiation and connection of multiple identical components such as half adders to make up a full adder, or *exclusive or* gates to create a parity tree.

IEEE VHDL LRM

Generate Statement FOR-Scheme

VHDL HOME PAGE

- All objects created are similar
- The GENERATE parameter must be discrete and is undefined outside the GENERATE statement
- Loop can not be terminated early

```
name : FOR n IN 1 TO 8 GENERATE
    concurrent-statements
END GENERATE name;
```

Generate Statement FOR-scheme

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

VHDL defines two different schemes for the GENERATE statement. These are the FOR-scheme and the IF-scheme. This slide shows the syntax for the FOR-scheme.

The FOR-scheme works in a similar manner as the FOR loop. The FOR-scheme generates the objects for a certain number of times and stops. In the FOR-scheme, all the objects are the same. The loop variable can be created in the GENERATE statement but it is undefined outside that statement.

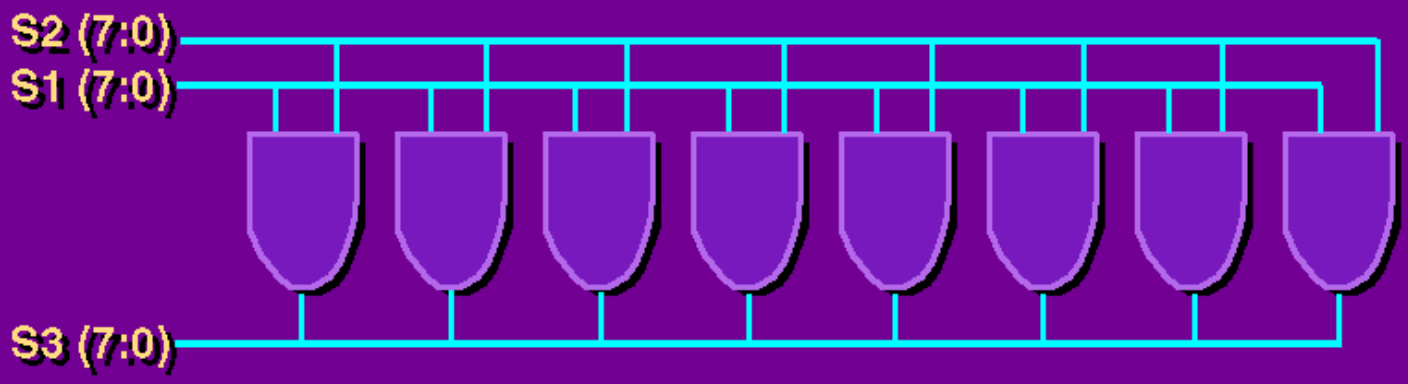
The syntax for the FOR-scheme GENERATE statement is shown in the slide. The loop variable in this case is N, but can be any valid VHDL identifier. The range can be any valid range, but must be discrete. After the GENERATE statement, the concurrent statements to be generated are stated. Finally, the GENERATE statement is closed by the END GENERATE construct.

IEEE VHDL LRM

FOR-Scheme Example

VHDL HOME PAGE

```
-- this uses the and_gate component from before
ARCHITECTURE test_generate OF test_entity IS
    SIGNAL S1, S2, S3 : BIT_VECTOR (7 DOWNTO 0);
BEGIN
   G1 : FOR n IN 7 DOWNTO 0 GENERATE
    and_array : and_gate
       GENERIC MAP (2 ns, 3 ns)
       PORT MAP (S1(n), S2(n), S3(n));
   END GENERATE G1;
END test_generate;
```



Generate Statement IF-Scheme

IEEE VHDL LRM

VHDL HOME PAGE

- Allows for conditional creation of components
- Can not use ELSE or ELSIF branches with the IF-Scheme

```
name : IF (boolean expression)GENERATE
    concurrent-statements
END GENERATE name;
```

IEEE VHDL LRM

Need for Configuration

VHDL HOME PAGE

- The configuration specification allows the designer to choose the entity for each component
- What is the need for configuration?
  - VHDL supports design partitioning
  - Various pieces of the design work may be parceled out
  - When the architecture is developed, only the component interface may be available
  - There is a need to pull the pieces of the design back together
- Configuration must account for
  - Entity name can be different than the component name
  - Entity declaration may have more ports than the component declaration
  - Ports on the entity declaration may have different names than the component declaration
- Configuration is clearly necessary in these cases to map the correct entity to the component

IEEE VHDL LRM

IF-Scheme Example

VHDL HOME PAGE

```
ARCHITECTURE test_generate OF test_entity IS
     SIGNAL S1, S2, S3 : BIT_VECTOR (7 DOWNTO 0);
BEGIN
   G1 : FOR N IN 7 DOWNTO 0 GENERATE

     G2 : IF (N=7) GENERATE
       or1 : or_gate
         GENERIC MAP (3ns, 3ns)
         PORT MAP (S1(n), S2(N), S3(N));
     END GENERATE G2;

     G3 : If (N < 7) GENERATE
       and_array : and_gate
         GENERIC MAP (2ns, 3ns)
         PORT MAP (S1(N), S2(N), S3(N));
     END GENERATE G3;

   END GENERATE G1
END test_generate;
```

Summary

- Structural VHDL describes the arrangement and interconnection of components
- Components can be of any level of abstraction - low level gates or high level blocks of logic
- Generics are inherited by every architecture or component of that entity
- Generate statements automatically create large, regular blocks of logic
- Configuration gives the designer control over the entity and architecture used for a component

Need for Configuration

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The configuration construct allows the designer to choose exactly what model a component will use. During the design process, various parts of the design can be parceled off to various teams or people. VHDL supports this "divide and conquer" strategy. However, once all the pieces have been developed, they must be brought together in a rational manner. Architectures may be developed without knowing the exact details of the components. Therefore, the exact component model must be chosen when compilation occurs. The configuration statement allows the designer to choose the component model appropriate for the design.

The configuration statement must account for many differences between the entity and the component. First, the entity name may be different than the component name. Second, the entity declaration may have more ports that the component declaration. A more generic part may have been developed with additional ports to keep the parts count down. However, these extra ports cannot be ignored. Finally, the ports of the entity may have different names than those on the component declaration. These different port names must be matched.

IEEE VHDL LRM

Configuration Specification

VHDL HOME PAGE

- The configuration specification must indicate two pieces of information
  - Selected components
  - Entity to bind with
- The basic syntax of the configuration specification

```
FOR component_specification USE binding_indication;
```

Configuration Specification

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The configuration must specify two pieces of information to the compiler. First, the effected components must be identified. Second, the entity to bind the information must be identified. In a configuration, the component is said to bind with an entity body. The basic syntax of the configuration specification is given in the slide. The specification must be given before any BEGIN statement in the architecture definition.

IEEE VHDL LRM

Component Specification

VHDL HOME PAGE

- The *component_specification* can be of several forms
  - Single component

```
For A1 : and_gate USE binding_specification;
```

  - Multiple components

```
FOR A1, A2 : and_gate USE binding_indication;
```

  - All components

```
FOR ALL : and_gate USE binding_indication;
```

All components of this type are affected

  - Other components

```
FOR OTHERS : and_gate USE binding_indication;
```

Components that have not yet been specified are affected

Configuration Specification: Example

- This example shows the use of the configuration specification to allow an entity to fit a component with a different interface

```
ENTITY JKFF IS
   PORT (clk, preset, clear, J, K : IN BIT;
         Q, Q_bar : OUT BIT);
END JKFF;

PACKAGE Global_signals IS
   SIGNAL clk, preset, clear : BIT;
END PACKAGE Global_signals;

USE Work.Global_signals;

ENTITY config_test IS
END config_test;
```

- The configuration statement maps the JKFF entity to the global signals and the ports of the component

```
ARCHITECTURE structural OF config_test IS
   SIGNAL S1, S2, S3, S4 : BIT;
   COMPONENT FF
     PORT (J, K: IN BIT; Q, Q_bar : OUT BIT);
   END COMPONENT FF;

   FOR UO : FF USE ENTITY Work.JKFF
     PORT MAP (clk => Global_signals.CLK,
       preset => Global_signals.preset,
       clear => Global_signals.clear,
        J => S1, K => S2,
        Q => S3, Q_bar => S4);

BEGIN
   UO : FF PORT MAP (S1, S2, S3, S4);
END structural;
```

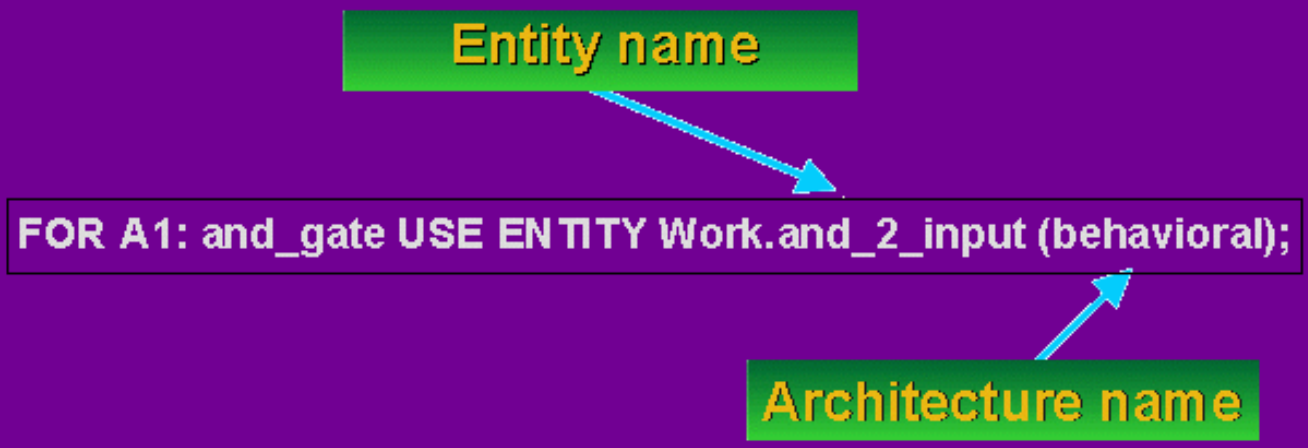
[Lipsett89]p. 137

IEEE VHDL LRM

Binding Indication

VHDL HOME PAGE

- The binding indication identifies the entity to be used for the component
- The name of the architecture can also be identified
- Configuration may also be made in a separate *configuration declaration*



- Binding indication may also include a PORT MAP and GENERIC MAP to adapt the entity to the component

Configuration Specification: Example

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This example shows how to use the configuration statement to bind an entity to a component.

First, entity *JKFF* is declared in the *Work* library. JKFF has five input ports and two output ports.

The package *Global_signals* declares signals that are available to all processes and components in the architecture.

The entity *config_test* is declared without input or output ports and can be thought of as the top level testbench in this example.

The architecture is declared with local signals *S1*, *S2*, *S3*, and *S4* and the component *FF*. The component *FF* has 2 inputs and 2 outputs. This component will be adapted to fit the *JKFF* component in a configuration specification.

The configuration specification first identifies the effected component. In this case, *U0*, the component instantiation of type *FF* is to be configured. The entity to be used is then stated (*JKFF* in the *Work* library). Since the interfaces do not match, a PORT MAP is necessary. The *clk*, *preset*, and *clear* signals of the entity are mapped to the signals declared in *Global_signals* with the same names. The *J*, *K*, *Q*, and *Q_bar* signals are mapped to the local signals.

Finally, the component instantiation statement maps the components to the same signals, completing the configuration.

References

References:

[Lipsett89] Lipsett, R., C. Schaefer, C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.

For further reading:

Bhasker, J., *A VHDL Primer*, Prentice Hall, 1995.

Calhoun, J.S., Reese, B., *Class Notes for EE-4993/6993: Special Topics in Electrical Engineering (VHDL)*, Mississippi State University, http://www.erc.msstate.edu/mpl/vhdl-class/html, 1995.

Coelho, D.R., *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993.

Navabi, Z., *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill, 1993.

Mentor Graphics Corporation, *An Introduction to Modeling in VHDL*, 1990.

Perry, D.L., *VHDL*, McGraw-Hill, 1994.

RASSP Roadmap

-- Notes Page --

This diagram emphasizes the role of VHDL in the RASSP program. VHDL can be used for system definition, functional design, hardware-software partitioning, hardware design and hardware-software integration and test. The concept of virtual prototyping uses VHDL as the binding language of choice for all design paradigms.

The most common usage of VHDL prior to RASSP was in the area of hardware design. The RASSP program has extended VHDL's use to include executable requirements, performance modeling/system level design as well as system integration and test.

Concepts of Structural VHDL

-- Notes Page --

Structural VHDL is concerned with the interconnection and arrangement of components describing the contents of a design. The behavior is not explicitly shown. It can be thought of as a physical netlist. A structural description need not be low level gates; it could represent the connections between high- level algorithmic elements of a system as well as low-level circuit elements of an ASIC design.

Rules for Actuals and Locals

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Locals are defined as the ports of the component. VHDL has two restrictions on the association of locals with actuals. First, the local and actual must be of the same data type. Second, the local and actual must be capable of being readable and/or writable. A local of mode IN can only be associated with an actual of mode IN. Ports of mode OUT behave similarly. A local INOUT port is generally associated with an INOUT or OUT actual.

Components from Packages

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This slide shows an example of the component declaration residing inside a package declaration. The package must be *included* in the file containing the architecture. This is done with the use of the *USE* statement. The *USE* statement can be read as follows. Use *ALL* components contained in the package *my_stuff* where the package *my_stuff* is contained in the library *work*. If the only element required from this package was *and_gate*, then the *USE* statement could have been written:

USE Work.my_stuff.and_gate

Also shown on this slide is the use of positional association. Positional association is used in the PORT MAP statement shown resulting in signals *S1*, *S2*, and *S3* being mapped to *in1*, *in2*, and *out1*, respectively. Association by name could also be used and the statement may have been:

PORT MAP (in1 => S1, out1 => S3, in2 => S2);

Note that when using association by name, the order of the signals in the *PORT MAP* is not important.

IEEE VHDL LRM

Generics

VHDL HOME PAGE

- Generics allow the component to be customized upon instantiation
- Generics pass information from the entity to the architecture
- Common uses of generics
  - Customize timing
  - Alter range of subtypes
  - Change size of arrays

IEEE VHDL LRM

Generic Map

VHDL HOME PAGE

- The GENERIC MAP is similar to the PORT MAP in that it maps specific values to generics declared in the component

```
PACKAGE my_stuff IS
   COMPONENT and_gate
      GENERIC (tplh, tphl : time);
      PORT (in1, in2 : IN BIT; out1 : OUT BIT);
   END COMPONENT;
END PACKAGE my_stuff;
```

```
USE Work.my_stuff.ALL;

ARCHITECTURE test of test_entity IS
     SIGNAL S1, S2, S3 : BIT;
BEGIN
  Gate1 : my_stuff.and_gate
     GENERIC MAP (2ns, 3 ns)
     PORT MAP (S1, S2, S3);
END test;
```

Restrictions on Instantiation

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Component instantiation must follow some rules to make sure the interface of the component and other objects match. To make discussion of this topic easier, the ports on the component are known as *locals*. Each *local* must match with an *actual*. An *actual* is a signal or formal port declared in the entity statement. As the picture above indicates, variables and constants cannot be associated with a port on a component; signals are the primary means of communication between VHDL entities and components.

IEEE VHDL LRM

IF-scheme Example

-- Notes Page --

VHDL HOME PAGE

The example here uses the IF-scheme GENERATE statement to make a modification to the *and_gate* array. The seventh gate of the array will be an *or_gate* when the GENERATE statement runs. In this example, the *or_gate* will need to have the same type of interface as the *and_gate*. The rest of the array is generated when the G3 generate block runs.

Another example use of the IF-scheme GENERATE is in the conditional execution of timing checks. Timing checks can be incorporated inside a GENERATE IF-scheme. For example, the following statement can be used:

Check_time : IF TimingChecksOn GENERATE

This allows the boolean variable *TimingChecksOn* to disable timing check simulation. This parameter can be set in a package or passed as a generic and can improve simulation speed by shutting off this computational section.

IEEE VHDL LRM

Generics

-- Notes Page --

VHDL HOME PAGE

Generics provide a method for information to be channeled from an entity (its input source) to a block (e.g. an architecture). It can be supplied either in a component instantiation or in a configuration specification or declaration. Generics can be used to pass timing information, control array widths and data sizes, or set particular control flags (e.g. turn on timing checks). A generic is much like a variable that is passed to a the component at the time of instantiation. Generic declarations may contain default values which can then be overridden during component instantiations.

IEEE VHDL LRM

Generics: An Example

VHDL HOME PAGE

- One use of generics is to alter the timing of a certain component
- It is possible to indicate a generic timing delay and then specify the exact delay at instantiation

```
COMPONENT and2 IS
   GENERIC (tplh, tphl : TIME);
   PORT ( input1, input2 : IN BIT;
          output : OUT BIT);
END COMPONENT;
```

- The example above declares the interface to the and2 component
- The propagation time for high-to-low and low-to-high transactions can be specified later

Generic Map

-- Notes Page --

Generics can be mapped in a fashion similar to ports. Generics are not required in a component but must be specified at instantiation if no default values are given in the instantiated component's *ENTITY*. In this example, the component *and_gate* is declared in the package *my_stuff*, and its generics are *tplh* and *tphl*. The values are assigned to the generics at instantiation in the *GENERIC MAP*. As in *PORT MAP* signal associations, associations may be made by position or by name.

FOR-scheme Example

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This slide shows an example of the FOR-scheme. While this is a simple example, far more complex designs can be generated.

The code generates an array of AND gates. The *and_gate* component used previously is employed here. In this case, the GENERATE statement has been named *G1* and is instantiating an array of 8 *and_gate* components. The PORT MAP statement maps the interface of each of the 8 gates to a specific element of the signal vectors. The leftmost *and_gate* has the seventh element of signals *S1*, *S2*, and *S3* mapped to its interface. This continues to the last *and_gate* of the array (rightmost) where the zeroth element of *S1*, *S2*, and *S3* is mapped to its interface.

Generate Statement IF-scheme

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The other form of the GENERATE statement is the IF-scheme. This scheme allows for conditional creation of objects. One obvious difference between this scheme and the FOR-scheme is that all the objects created do not have to be the same. While this IF statement may seem similar to the IF-THEN-ELSE construct in behavioral VHDL, the IF-scheme does not allow the use of ELSE or ELSIF branching.

The syntax for the IF-scheme GENERATE statement is shown in this slide. The boolean expression of the IF statement can be any valid boolean expression.

Generics: An Example

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This slide gives an example of how generics may be used. In this case, the component *and2* has two generic parameters associated with its entity. They are *tplh* and *tphl* and represent the propagation time for low-to-high and high-to-low transitions.

Note that since no default values are assigned here, the actual values of these generics must be set when the component is instantiated or when a configuration specification is done.

Component Specification

-- Notes Page --

The *component specification* can be of several forms. This slide shows examples for various types. The *component specification* identifies those components to be configured. Both single and multiple components can be selected. The keyword ALL selects all components of that type. The keyword OTHERS selects all components not yet configured. The OTHERS keyword is similar to a default value.

Binding Indication

IEEE VHDL LRM

-- Notes Page --

VHDL HOME PAGE

The *binding indication* is the second part of the configuration specification. It identifies the entity to bind with the component and also maps the two interfaces together. That is, a *binding indication* associates component instances with a particular design entity. A particular architecture of the entity can also be selected at this time. The binding indication may include a PORT MAP and GENERIC MAP to adapt the interfaces of the entity and the component.

An alternate method of binding components is to use a configuration declaration. This is a separate library that specifies the configuration data for an architecture. This method is mainly used for large scale design and will not be discussed in this presentation.

Behavioral VHDL - Module 3

Table of Contents

IEEE VHDL LRM

VHDL HOME PAGE

Behavioral VHDL - Module 3

This module was prepared as part of the RASSP Education & Facilitation effort.

Version 1.0

Toolbar Functionality

	Takes the user up one hierarchicall level in the presentation.
	Takes the user to the previous section of the presentation.
	Takes the user to the previous slide in the presentation.
	Takes the user to a listing of all slides with links to each slide.
	Takes the user to a visual representation of the organization of the slide presentation.
	Takes the user to a document, further explaining the information contained within the slide.
	Brings the user to this document, containing information on the use of the toolbar.
	Takes the user to the next slide in the presentation.
	Takes the user to the next section of the presentation.
	Takes the user down one hierarchical level in the presentation.

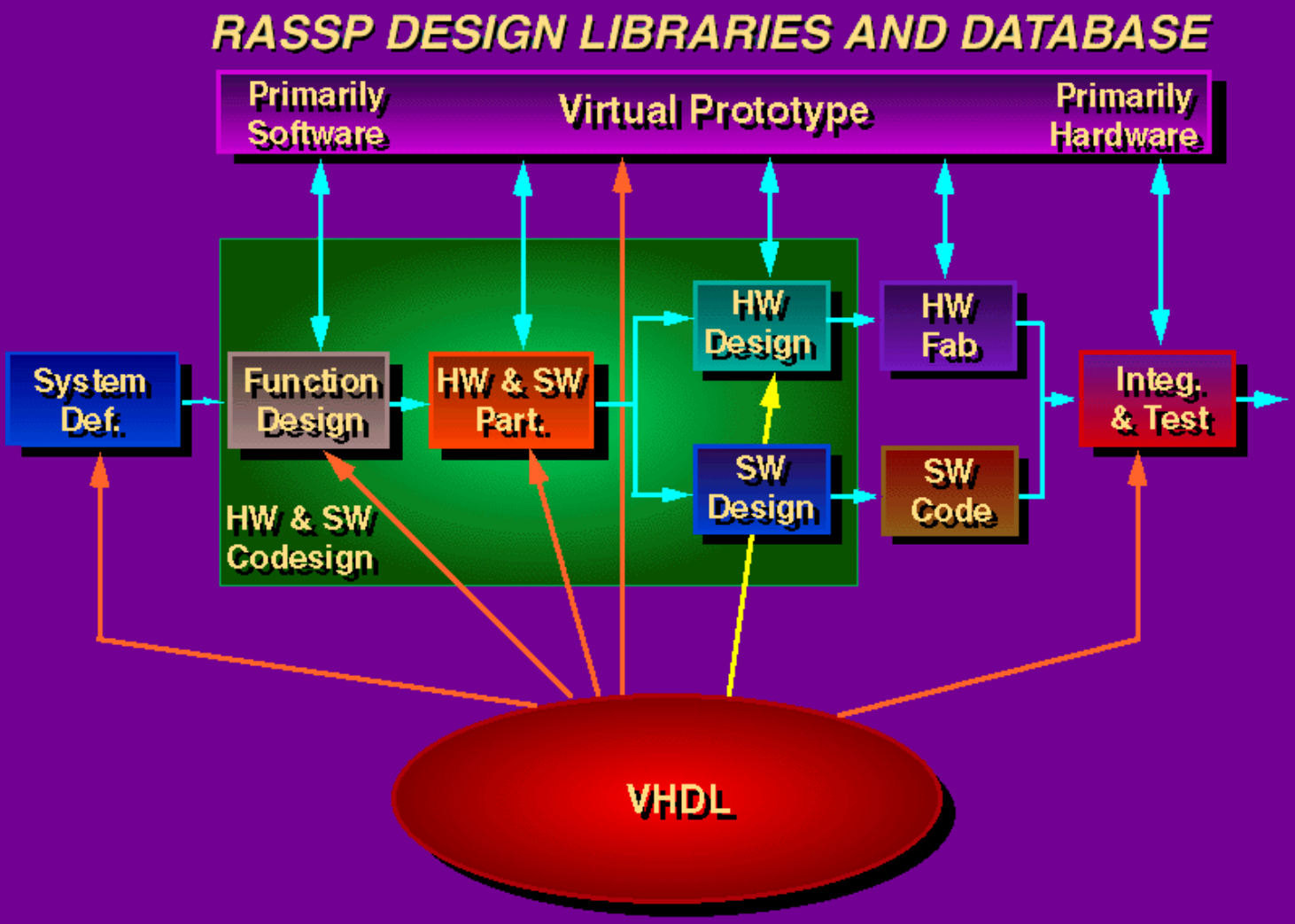
Behavioral VHDL - Module 3

Outline

IEEE VHDL LRM

RASSP Roadmap

VHDL HOME PAGE



Behavioral VHDL - Module 3

Module Goals

- Comprehension of behavioral VHDL constructs
- Expansion of knowledge of VHDL concepts and syntax
- Understanding of the application of behavioral VHDL to a real example

Introduction to Behavioral Modeling in VHDL

- Abstraction levels of VHDL models
  - Structural level
  - Behavioral/structural mixed (i.e., data flow)
  - Behavioral
- Behavioral Modeling
  - Functional performance is the goal of behavioral modeling
  - Timing optionally included in the model
  - Software engineering practices should be used to develop behavioral models
    - Structured design
    - Iterative refinement
    - Abstract data typing
    - Loose coupling, strong cohesion

Introduction to Behavioral Modeling in VHDL

-- Notes Page --

Using VHDL, a system designer can model a circuit (i.e., a component or system) at multiple levels of abstraction. In prior lessons, we have concentrated on the basic elements and the structural forms of describing models in VHDL. In this module we concentrate on the behavioral view, that is, describing how the circuit is to perform.

We hide the structure of the design when modeling a circuit behaviorally. Instead, we are vitally interested in the functionality of the circuit. At the highest levels of abstraction, we even ignore timing.

When modeling in VHDL it is important to follow standard practices of software engineering. Otherwise, the model will be hard to maintain, even by the person who wrote it. In addition, to aid the reuse of models, even "throw-away" models should be created with care, and with the thought that others may use it.

Typical model design and coding practices include structuring the design, iteratively refining a high-level view of the model down to its final form, employing abstract data typing to hide and encapsulate data, and organizing the individual model components so that they are loosely coupled (small number of interface signals) and have strong cohesion

(keep strongly related functions in the same architectural body).

IEEE VHDL LRM

Example Behavioral VHDL Model

VHDL HOME PAGE

```
USE TEXTIO.all, mypackage.all;

ENTITY module is
  PORT (X, Y: in BIT; Z: out BIT_VECTOR(3 DOWNTO 0);
END module;

ARCHITECTURE behavior OF module is
  SIGNAL A, B: BIT_VECTOR(3 DOWNTO 0);
BEGIN
  A(0) <= X AFTER 20 ns; A(1) <= Y AFTER 40 ns
  PROCESS (A)
       VARIABLE P, Q: BIT_VECTOR(3 DOWNTO 0);
  BEGIN
       P := fft(A);
       B <= P AFTER 10 ns;
  END PROCESS;
  Z <= B;
END behavior;
```

Example Behavioral VHDL Model

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Here, we see an example of behavioral VHDL which includes the use of VHDL signals as well as variables. A process is shown which includes a series of statements executed sequentially. The process itself, however, is executed concurrently with the assignments to *A* and *Z*. The ability to model both concurrent and sequential events in a VHDL model will be covered in detail in subsequent sections of this module.

Note that variables are also shown; these are unique to behavioral VHDL in that they can only be used inside processes. It is important to point out that this material will not cover shared or global variables added in the VHDL 93 revisions.

The functionality of the component is defined in the architecture body. Two signals, *A* and *B* are defined internal to the component. Three statements are defined:

A(0) <= *X* AFTER 20 ns;
A(1) <= *Y* AFTER 40 ns;
PROCESS (*A*);

The process executes the Fourier transform of *A* using a function and transfers the results to signal *B*. All three statements execute concurrently. The two signal assignment statements are activated whenever a signal in their respective right-hand sides changes value. The process becomes

active when there is a change in *A*.

IEEE VHDL LRM

VHDL Processes

VHDL HOME PAGE

- A VHDL process statement is used for all behavioral descriptions
- Example simple VHDL process

```
ARCHITECTURE behavioral OF clock_component IS
BEGIN
    PROCESS
      VARIABLE periodic: BIT := '1';
        BEGIN
         IF en = '1' THEN
            periodic := not periodic;
         END IF;
         ck <= periodic;
         WAIT FOR 1 us;
    END PROCESS;
END behavioral;
```

Case Study of the SDSP Microprocessor Organization

- Simple 32-bit microprocessor with
  - 32-bit address and data bus
  - 256-word register file
  - 3-operand addressing
  - "Quick" mode for arithmetic and I/O instructions
- On reset, the SDSP PC initialized to zero; all other regs undefined
- By convention, R0 holds zero - but must be set by software
- Condition codes:
  - V: overflow (set by arithmetic result larger that can be represented)
  - N: negative (set if arithmetic result is negative)
  - Z: zero (set if arithmetic or logic result is zero)

Resolving Difficulties

- Overloaded items cannot be resolved if the argument types include common literals. i.e.,

```
 TYPE twobit IS ('0', '1');
TYPE fourbit IS ('U', '0', '1', 'Z');
FUNCTION abc (x: twobit) RETURN INTEGER;
FUNCTION abc (x: fourbit) RETURN INTEGER;
    ....
y <= abc('0');          --Which function do we use?
```

  - Resolve the ambiguity by qualifying the literal:

```
  y <= abc(twobit'('0');
```

  - General tip: Use qualification to avoid numerous problems where the compiler cannot seem to select a specific meaning, e.g., *read (abc, string'("abcabc"))*;

SDSP Microprocessor Organization

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The SDSP processor is a full 32-bit system (both data and instructions). It is organized along the lines of a RISC architecture. For example, every instruction is exactly 32-bits in length (plus 32-bit displacement if needed).

To keep the SDSP simple there are a few capabilities missing. For example, the SDSP has no interrupt capability, nor does it have subroutine support. But due to the nature of the model, these items can be very easily and quickly added. The three condition code register bits are updated after each arithmetic or logical instruction.

Z - zero bit
    set if the result is zero
N - negative bit
    set if the result of an arithmetic instruction is negative
V - overflow bit
    set if a carry or borrow is created from MSB or LSB, respectively

The PC is set to zero on reset, and the values in the other registers are not specified. By convention, R0 is read-only and always contains zero (not enforced by hardware). The PC is incremented to point to the next address after each instruction is fetched.

IEEE VHDL LRM

SDSP Microprocessor Instruction Architecture

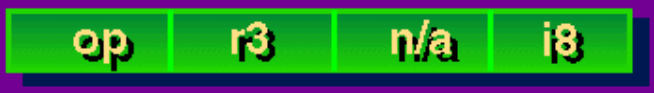
VHDL HOME PAGE

- Arithmetic and Logical (add, sub, mul, div, addq, subq, mulq, divq, land, lor, lxor, lmask)

8-bit fields

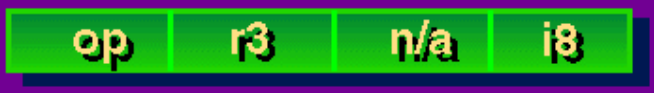
r3 <- r1 op r2
r3 <- r1 op i8

- I/O (ld, st, ldq, stq)



r3 <- r3 shifted by i8

- Shift (shl, shr, shla, shra, rotl, rotr)



r3 <- r3 shifted by i8

- Branch (br*, brq*, bi*, biq*: where *= v, nv, nn, p, np, z, nz)

br*

cond=((V&v)|(N&n)|(Z&z))=i

brq*

bi*

biq*


Map
Notes
Help

Summary

- Behavioral VHDL is used to focus on the behavior, and not the structure, of the device
- Several familiar programming constructs, such as CASE and IF-THEN-ELSE statements, are available
- Subprograms allow large parts of code to be broken down into smaller, more manageable parts
- Bus resolution functions decide the final value of multiple signal assignments to one signal

VHDL Processes

-- Notes Page --

We now turn our attention to a the VHDL process statement. The process is the key structure in behavioral VHDL modeling. A process is the only means by which the executable functionality of a component is defined. In fact, for a model to be capable of being simulated, all components in the model must be defined using one or more processes.

Statements within a process are executed sequentially (although care needs to be used in signal assignment statements since they do not take effect immediately; this was covered in the VHDL Basics module when the VHDL timing model was discussed). Variables are used as internal place holders which take on their assigned values immediately.

All processes within an architecture body (or within a whole model for that matter) are executed concurrently. That is, although statements within a process are evaluated and executed sequentially, all processes within the model begin executing concurrently.

In the example process given here, a variable *periodic* is declared and assigned the initial condition '1'. As long as *en* is '1', if *periodic* changes value, then a future possible changed value (called a transaction) is created and saved on a list of transactions for *ck* by the simulator. The process then suspends for one microsecond. The signal *ck* actually assumes its new value one delta cycle after the process suspends (because the transaction becomes an event). After the one microsecond suspension, the process once again executes beginning with the *if* statement. Note that only variables can be declared in a process, and signals (which are global

to a process) are used primarily as control (e.g., *en* in this case), inputs into a process, or outputs from a process (e.g., *ck* in this case).

IEEE VHDL LRM

Process Syntax

VHDL HOME PAGE

```
[process_label :] PROCESS
[(sensitivity_list)]

  process_declarations

BEGIN

  process_statements

END PROCESS [process_label];
```

VHDL Sequential Statements

- Assignments executed sequentially in processes
- Sequential statements
  - {Signal, variable} assignments
  - Flow control
    - if <condition> then <statements> else <statements> end if;
    - for <range> loop <statements> end loop;
    - while <condition> loop <statements> end loop;
    - case <condition> is when <value> => <statements>;
      when <value> => <statements>;
      when others => <statements>;
      end case;
  - Wait on <signal> until <expression> for <time>;
  - Assert <condition> report <string> severity <level>;

IEEE VHDL LRM

VHDL Packages

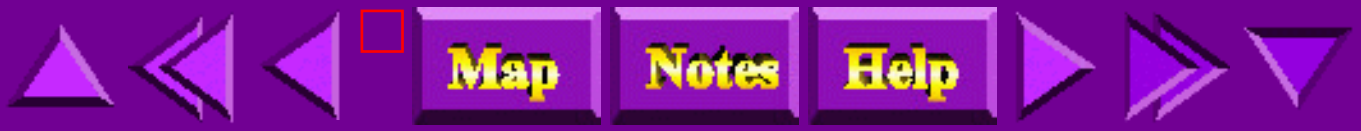
VHDL HOME PAGE

- What can you put in a package?
  - Subprograms (i.e., functions and procedures)
  - Data and type declarations such as:
    - User record definition
    - User types and enumerated types
    - Constants
    - Files
    - Aliases
    - Attributes
  - Component declarations
- Entities and Architectures *cannot* be declared or defined in a package
- To use a package, make it visible via the "use" language construct

Potential Problems to Avoid (cont. 1)

- Avoid using shared variables
  - Debugging potential asynchronous errors can be difficult
  - Concept likely to change in future VHDL standards
- Overloaded items cannot be resolved by return type
  - Example: These overloaded functions cannot be disambiguated:

```
FUNCTION "-" (a, b: natural) RETURN integer;
FUNCTION "-" (a, b: natural) RETURN natural;
```



Resolving Difficulties

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Because literals in VHDL are semantically ambiguous (e.g. "abc" can be a string or a vector of enumerated values 'a', 'b', 'c'), it is often impossible for the VHDL analyzer to determine the exact type of a literal, and thus resolve the overloaded function, if it is dependent on the literal.

For instance, note that in the upper example, '0' appears in the definition for both enumerated types, *twobit* and *fourbit*. Therefore, calling *abc* with '0' as its parameter does not allow for a distinction between the two versions of the *abc* function.

It is a good idea to use qualification when passing literals as subprogram parameters both to ensure that inadvertent ambiguities are avoided and to improve the readability of the VHDL code.

DSP Microprocessor Instruction Architecture

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The instruction mnemonics ending with 'q' are *quick* instructions. That is, the second operand is the value of the LSB of the instruction rather than in r2 or a 32-bit displacement immediately following the instruction.

The shift instruction fills in with 0 for non-arithmetic shifts, and the sign bit for arithmetic shifts. "n/a" means not applicable, and the contents of that field are ignored.

Branch instructions are of four types:

br* -
simple branch

brq* -
simple branch where displacement is low-order eight bits of instruction

bi* -
branch indexed where branch address is pc+displacement+[r1]

biq* -
branch indexed where displacement is low-order eight bits of instruction

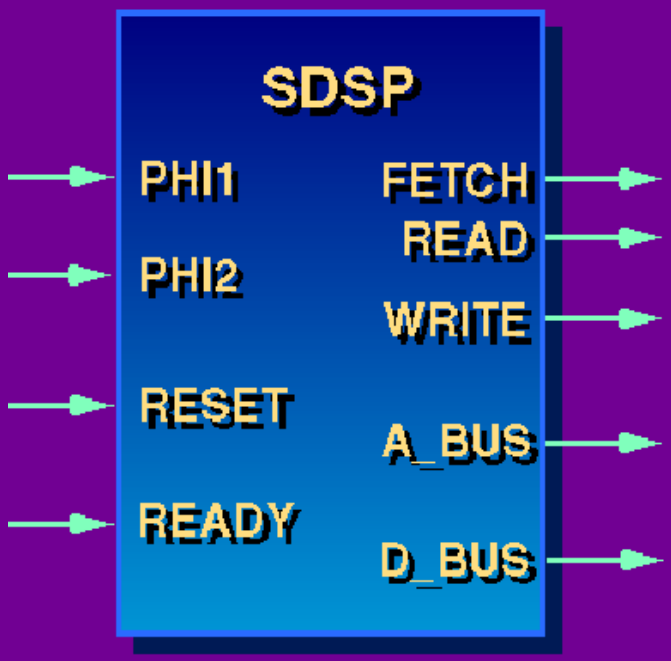
---

IEEE VHDL LRM

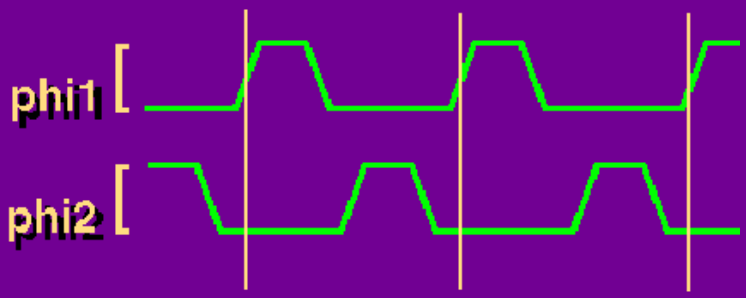
SDSP Context and Clock

VHDL HOME PAGE

Entity



Two-phase non-overlapping clock



Map Notes Help

SDSP Benchmark


```
        mul r9 r1 r7    ; r7 = x mod q
        mul r8 r4 r6
        sub r8 r9 r8
        addq r5 r5 0    ; x = r9+m if r9 > 0
        brpq next
        add r5 r5 r2
        stq r5 r0 x     ; save new seed in mem[x]
next :  div r5 r5 r10   ; r5 = r5/scale
        stq r5 r0 result        ; save random result in
mem[result]
   ;
   ;------- End of generator algorithm
   ;
        st r5 r9 list      ; numbers stored starting from end of
list
        brq again
   ;
   ;------- All 100 numbers generated, so quit
   ;
        done :    stq r5 r0 donetrigger ; monitor address bus for
"donetrigger" to stop timing
        idle :    brq idle              ; busy loop
             end
```

References

For further reading:

IEEE Stadnard VHDL Language Reference Manual, IEEE Std. 1076-1993.

Ashenden, Peter, *The VHDL Cookbook*, 1989 (unpublished). Available via ftp from thor.ece.uc.edu.

Jain, Ravi, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991.

Mohanty, S., Krishnaswamy, V., and Wilsey, P., "System Modeling, Performance Analysis, and Evolutionary Prototyping with Hardware Description Languages", *Proceedings of the 1995 Multiconference on Simulation*, pp 312-318.

Navabi, Z., *VHDL Analysis and Modeling of Digital Systems*, McGraw-Hill, 1993.

Process Syntax

IEEE VHDL LRM

-- Notes Page --

VHDL HOME PAGE

The use of "process_label" at the beginning and end of a process is optional but recommended to enhance code readability.

The "sensitivity_list" is optional in that a process may have either a sensitivity_list, or it must include "wait" statements. A process cannot include both a sensitivity_list and wait statements. Wait statements will be covered in a subsequent section.

The "process_declaration" includes declarations for variables, constants, aliases, files, and many other things.

The "process_statements" include variable assignment statements, signal assignment statements, procedure calls, wait statements, "if" clauses, "while" loops, assertion statements, etc.

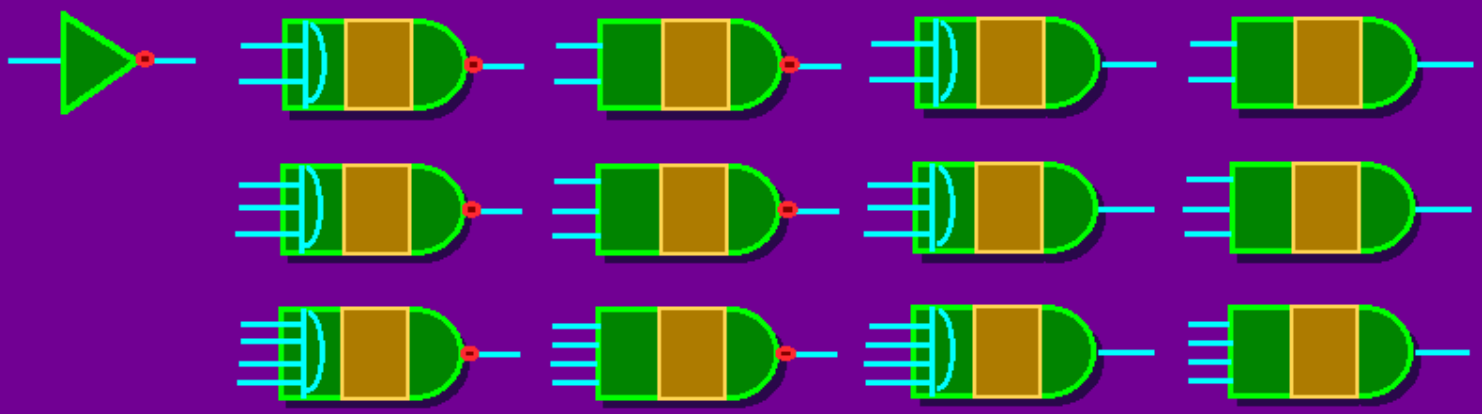
IEEE VHDL LRM

Let's write a VHDL Model

VHDL HOME PAGE


```
ENTITY full_adder IS
      PORT (A, B, Cin: IN bit;
         Sum, Cout: OUT bit);
END full_adder;
```

Can we build the Full Adder's architecture using these gates?



IEEE VHDL LRM

Alternate Carry Process

VHDL HOME PAGE

```
Carry: PROCESS(A, B, Cin) BEGIN IF (A = '1' and B
= '1') THEN Cout <= '1'; ELSIF (A = '1' and Cin =
'1') THEN Cout <= '1'; ELSIF (B = '1' and Cin =
'1') THEN Cout <= '1'; ELSE Cout <= '0'; END IF;
END PROCESS Carry;
```

VHDL Sequential Statements

-- Notes Page --

Sequential statements are used within processes and are executed in a top-down fashion. The list shown on this page includes many of the commonly used forms, but the list is not complete. The VHDL Language Reference Manual (IEEE Std. 1076-1993) provides a complete list.

Each of these statement types will be explained in further sections of this module. Some of you may note that these control structures operate almost exactly like their counterparts in Ada except for the assert and sequential signal assignment statements.

IEEE VHDL LRM

A Design Example - 2-bit Counter

VHDL HOME PAGE

```
ENTITY count2 IS GENERIC (prop_delay : TIME :=
10ns); PORT (clock : IN BIT; q1, q0: OUT BIT); END
count2; ARCHITECTURE behavior OF count2 IS BEGIN
count_up: PROCESS (clock) VARIABLE count_value:
NATURAL := 0; BEGIN IF clock='1' THEN count_value
:= (count_value+1) MOD 4; q0 <=
bit'val(count_value MOD 2) AFTER prop_delay; q1 <=
bit'val(count_value/2) AFTER prop_delay; END IF;
END PROCESS count_up; END behavior;
```

IEEE VHDL LRM

Blocks and Guards

VHDL HOME PAGE

- Blocks partition the concurrent statements in an architecture such that conditional activities unique to each block can occur
- A guarded signal assignment statement generates a value only if the block guard expression is true. If false, the assignment is *disconnected*
- Example

```
ARCHITECTURE guarded_assignments OF n_1_mux IS
BEGIN
  bi: FOR j IN i'range GENERATE
    bj: BLOCK (s(j) = '1' or s(j) = 'z')
        BEGIN
         x<= guarded i(j);
        END BLOCK;
  END GENERATE;
END guarded_assignments;
```

Potential Problems to Avoid

- Objects defined by subtypes derived from a base type are considered being of the same type
  - Example:

```
PROCESS
  SUBTYPE smallintA IS integer (RANGE 0 to 10);
      SUBTYPE smallintB IS integer (RANGE 0 to 15);
      VARIABLE A: smallintA := 5;
      VARIABLE B: smallintB := 8;
      VARIABLE C: integer;
      BEGIN
          B := B * A;       --OK
          C := B + 1;       --OK
 END PROCESS;
```

Potential Problems to Avoid (cont. 1)

-- Notes Page --

The use of shared variables requires careful attention to ensure that correct values are communicated among relevant processes. For example, if one process writes a shared variable in the same simulation cycle that another process reads the variable, the VHDL standard does not define what value is read. Similarly, if two or more processes write to the same shared variable in the same simulation cycle, the standard does not define what value should be written to the variable.

Care must be taken if overloaded functions are differentiated solely by the type of their return values. The previous version of the VHDL standard, 1076-1987, did not require that differentiations on output types be supported. The current standard, 1076-1993, however, has included the requirement that differentiations based solely on output type be supported.

SDSP Context and Clock

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The signals shown in the entity are all single-bit except for the address bus, *A_BUS*, and the data bus, *D_BUS*, which are 32-bits each.

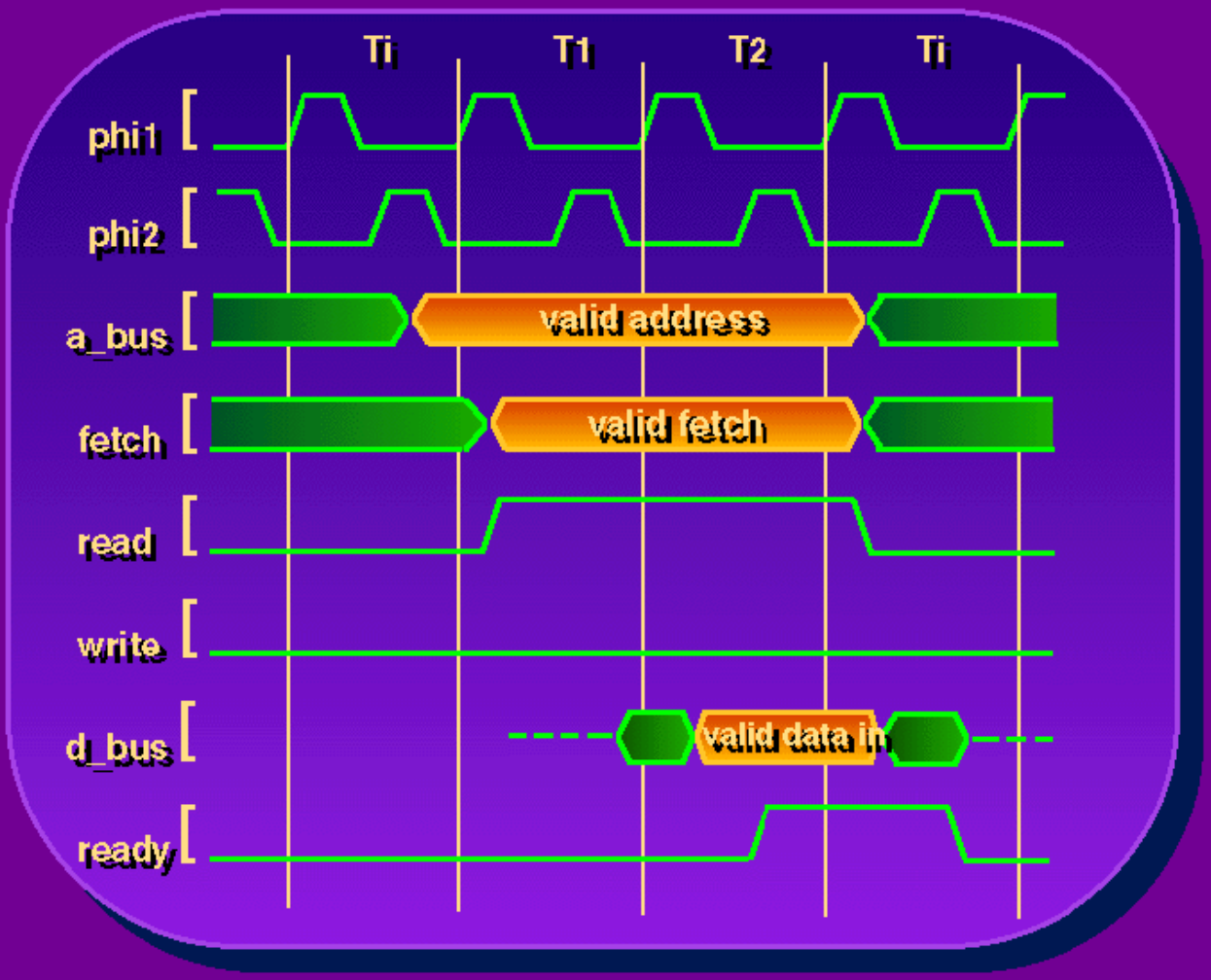
The clock is a two-phase clock with non-overlapping phases. Each cycle of *phi1* defines a bus state of which there are three: Ti, T1 and T2. Ti is the idle state.

A bus transaction (e.g., read or write memory) consists of a T1 state followed by one or more T2 states. The *fetch* port is a status signal indicating an instruction fetch is in progress. The *ready* port is set by the memory to indicate that read data is available or write data has been accepted.

IEEE VHDL LRM

SDSP Bus Read Timing

VHDL HOME PAGE



* Three cycles minimum to read data or instruction

* 1st, assert address bus

* 2nd, assert read and wait for ready from memory to go high

* 3rd, strobe data into CPU on leading edge of ready

SDSP Benchmark


```
   ;
    ;---------- Generate 100 random numbers
    ;
                 ldq r9 r0 counter      ; r9 is loop ctr
    again :      subq r9 r9 1          ; decrement counter
                 brnq done             ; jump out when all 100 #'s
generated
    ;
    ;--- The following code is the random number generator
algorithm
    ;
    ;       x_new = a * x_old * mod m
    ;       result = x_new/divisor
    ;
    ; Ensure seed is in memory at loc "x" if it isn't already there
from the previous execution
    ; of the algorithm
    ;
                  ldq r5 r0 x      ; put new seed in reg 5
                  div r6 r5 r3     ; r6 = x div q
                  mul r7 r6 r3
                  sub r7 r5 r7
```

Let's write a VHDL model

-- Notes Page --

As was seen in an earlier module, a VHDL model contains an entity and an architecture. Here the entity, which defines the model's interface to the outside world, is shown.

Full Adder Architecture

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

for Carry:

		Cin	
A	B	0	1
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

for Sum:

		Cin	
A	B	0	1
0	0	0	1
0	1	1	0
1	1	0	1
1	0	1	0

IEEE VHDL LRM

Complete Architecture

VHDL HOME PAGE

```
ARCHITECTURE example OF Full_Adder IS
    --Nothing needed in declarative block...
BEGIN

 Summation: PROCESS(A, B, Cin)
     BEGIN
         Sum <= A xor B xor Cin;
 END PROCESS Summation;

 Carry: PROCESS(A, B, Cin)
     BEGIN
         Cout <=  (A and B) or
                  (A and Cin) or
                  (B and Cin);
 END PROCESS Carry;

END Example;
```

Alternate Carry Process

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The Carry output could have been described using programming language constructs instead of the logic equations shown previously. Here, a set of nested if-then-else statements is used to implement the table lookup method. A case statement could also be used.

A Design Example - 2-bit Counter

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

In this example, we show a model for a simple 2-bit counter which counts clock pulses. The component has *clock* as an input, and two outputs which represent the LSB and MSB of a two-bit unsigned number.

There are several constructs that you may not have seen before. *bit'val(count_value mod 2)* is a function which returns a value of type bit. *count_value* is a natural number (i.e. an integer greater than, or equal to, zero). *count_value mod 2* returns the LSB value of the counter value, but the LSB value is of type natural. Since we want the LSB to be of type bit instead, we cast it by using the *'val* (read as "tic val") attribute on the type bit.

Another possibly new concept shown here is *generic*. A generic is like a port except it is treated as a constant in the architecture. Thus, *prop_delay* is a constant which can be set by the modeler when the entity *count2* is instantiated as a component. If the modeler decides not to explicitly set the value of *prop_delay*, its value defaults to 10 nanoseconds.

IEEE VHDL LRM

The Wait Statement

VHDL HOME PAGE

- The wait statement causes the suspension of a process statement or a procedure
- wait [sensitivity_clause] [condition_clause] [timeout_clause];
  - sensitivity_clause ::= on signal_name {, signal_name}

```
wait on CLOCK;
```

  - condition_clause ::= until boolean_expression

```
wait until Clock = '1';
```

  - timeout_clause ::= for time_expression

```
wait for 150 ns;
```

IEEE VHDL LRM

Entity Statements

VHDL HOME PAGE

- Entities may contain statements (including processes and procedures), but the statement can only be
  - Concurrent assertion statements
  - *Passive concurrent procedure calls*
  - *Passive process statements*
- Example

```
ENTITY multiplexor IS
        PORT (a, b: IN BIT; select: IN BIT; output: OUT BIT);

BEGIN
      check: PROCESS(a, b)
      BEGIN
            ASSERT a/=b REPORT "a equals b" SEVERITY NOTE;
      END PROCESS;
END multiplexor;
```

Blocks and Guards

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Blocks are used both to define a hierarchy within a design and to group together signal assignments which are only to be evaluated, and, in fact potentially connected, when the defined GUARD is true.

The primary reason to organize an architecture as several blocks of activity is to control the actions of groups of concurrent statements.

A conditional GUARD can be stated at the head of a block. If such a condition expression exists, then any statement in the block which has the keyword *guarded* will disconnect if the expression evaluates to false.

Potential Problems to Avoid

-- Notes Page --

Interestingly, even thought VHDL is considered to be strongly typed, the developers of the language decided to strongly type only with respect to the base type, not derived subtypes.

Thus, the VHDL analyzer will not be aware of inconsistent subtypes in the example shown here, and the simulator will execute the statements as expected. Note, however, that the result after multiplying *A* and *B* may be out of the range of *B*'s subtype.

SDSP Bus Read Timing

IEEE VHDL LRM

VHDL HOME PAGE

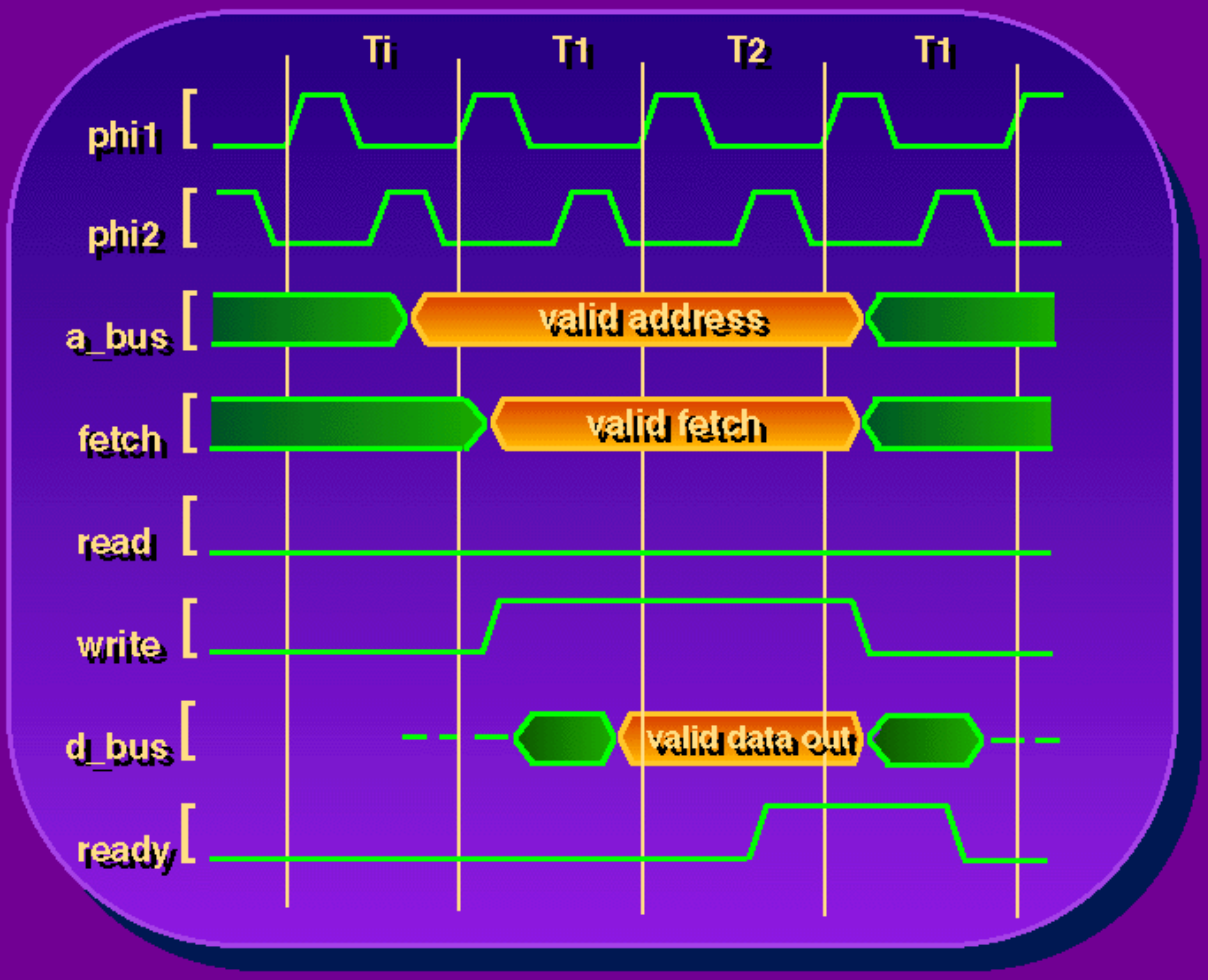
-- Notes Page --

1. During a Ti state, the CPU places an address on *A_BUS*. T1 is the next state
2. After the leading edge of *phi1*, the CPU asserts *read* to initiate a read activity in the memory
3. If an instruction is being fetched, *fetch* is asserted
4. T2 states occur until *ready* is asserted by the memory
5. The CPU inputs data on rising edge of *phi1* and deasserts *read* (and *fetch*).
6. Memory deasserts *ready* on falling edge of *read*

IEEE VHDL LRM

SDSP Bus Write Timing

VHDL HOME PAGE



* Three cycles minimum to write data

* 1st, assert address bus and data bus

* 2nd, assert write

* 3rd, data written when ready asserted

Map Notes Help

IEEE VHDL LRM

SDSP Benchmark

VHDL HOME PAGE

```
   ;
    ;----------- constants and data areas for the random generator
    ;
      a : data 16807
      m : data 2147483647     ; (2**31)-1
      q : data 127773         ; m div a
      r : data 2836           ; m mod a
    scale : data 10000000     ; scale factor to produce random
number between 0 and
    214
      x :   data 20           ; starting seed (updated each time
algorithm executes)
    result : res 1            ; random result between 1 and 214
    ;
    ;---------- load constants into registers for the generator
algorithm
    ;
    begin :  ldq r1 r0 a
             ldq r2 r0 m
             ldq r3 r0 q
             ldq r4 r0 r
             ldq r10 r0 scale
```

SDSP Benchmark


```
   ;
    ;   Random number generator including code to test the
generator
    ;
    ;
    ;   Hal Carter, 14 Feb 95 Ver. 1.0
    ;   This random number generator is a integer linear-
congruential generator with period
    ;   (2**31)-1 as presented in R. Jain, "The Art of Computer
Systems Performance
    ;   Analysis, " John Wiley & Sons, pp.  441-444, 1991.
    ;-------------------------------------------------
                org 0
                lmask r0 r0 r0              ; R0 <- 0
                stq  r0 r0 starttrigger  ;monitor address start to
start timing monitor
                brg begin                 ; jump around data segment
    ;
    ;--------- constants and data areas for
    ;--------- testing the generator
    ;
           list : res 100                 ; list of random numbers
        counter : data 100
     starttrigger : res 1
    donetrigger : res 1
```

Full Adder Architecture

IEEE VHDL LRM

-- Notes Page --

VHDL HOME PAGE

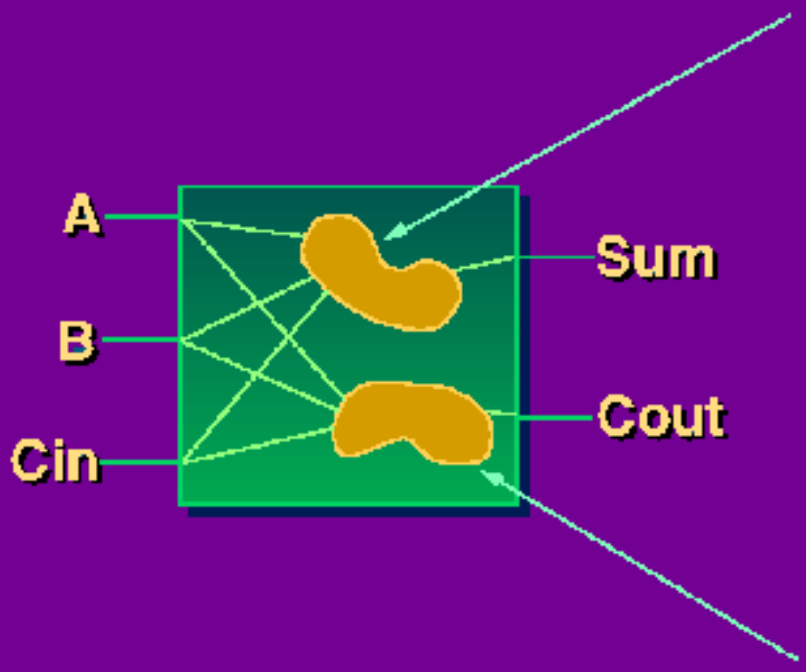
A one-bit full adder will be used in the next few pages as an ongoing example.

One way to describe the function of a full-bit adder is as a look-up table. In other words, we can define every mapping of the inputs to the outputs, and encode them as a case statement in the body of the process. Here, the logic tables used to generate the outputs, *Sum* and *Cout*, are shown.

IEEE VHDL LRM

Two Full Adder Processes

VHDL HOME PAGE




```
Summation:
PROCESS(A,
B, Cin)
BEGIN Sum
<= A xor B
xor Cin;
END PROCESS
Summation;
Carry
PROCESS(A,
B, Cin)
BEGIN Cout
<= (A and
B) or (A
and Cin) or
(B and
Cin); END
```

```
PROCESS
Carry;
```

Complete Architecture

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Now we put the entire architecture together. The two processes defined on the previous slide are placed in the same architecture. Note that the *SUM* and *CARRY* processes execute concurrently.

This model does not exploit explicit time (that is, there are no AFTER phrases or "wait for" statements. Thus, this model is purely functional. If timing is important, we can either add delay phrases (i.e., AFTER clauses) to the signal assignment statements, or use "wait for" statements.

Note that if wait statements are used in a process, the process cannot have a sensitivity list.

The Wait Statement

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Wait statements are used to suspend the execution of a process until some condition is satisfied. Processes in VHDL are actually code loops. The execution of the last statement in the process is followed by the execution of the first statement in the process, and process execution continues until a wait statement is reached. For this reason, every process must have at least one wait statement (a sensitivity is actually an implied wait statement which will be described in the next page of this module).

The structure of a wait statement contains optional clauses which can be used in any combination:

The sensitivity_clause:

the wait statement will only evaluate its condition clause when there is an event (i.e. a change in value) on at least one of the signals listed in the sensitivity_clause. If no sensitivity_clause is given, the signals listed in the condition_clause constitute an implied sensitivity_clause.

The condition_clause:

an expression which must evaluate to TRUE in order for the process to proceed past the wait statement. If no condition_clause is given, a TRUE value is implied when there is an event on a signal in the sensitivity_clause.

The timeout_clause:

specifies the maximum amount of time the process will remain suspended at this wait statement. If no timeout_clause is given,

STD.STANDARD.TIME'HIGH-STD.STANDARD.NOW (effectively until the end of simulation time) is assumed.

Wait statements assist the modeling process by synchronizing concurrently executing processes, implementing delay conditions into a behavioral model, or establishing event communications between processes. Sometimes, wait statements are used to sequence process execution relative to the simulation cycle.

Equivalent Processes

- "Sensitivity List" vs "wait on"

```
Summation:
  PROCESS(A, B, Cin)
    BEGIN
      Sum <= A xor B xor Cin;
END PROCESS Summation;
```

=

```
Summation: PROCESS
    BEGIN
      Sum <= A xor B xor Cin;
         WAIT ON A, B, Cin;
END PROCESS Summation;
```

if you put a sensitivity list in a process, you can't have a wait statement!

if you put a wait statement in a process, you can't have a sensitivity list!

IEEE VHDL LRM

Null Transactions

VHDL HOME PAGE

- How can a driver be disconnected (i.e., not influencing the output at all)?
  - Use the null waveform element
- Example:

```
bus_out <= null after 17 ns;
```

- What happens if all drivers of a resolved signal are disconnected?
  - Use register kind in signal declaration to keep most recently determined value
  - Use bus kind in signal declaration if resolution function must determine the value
  - Example:
    - signal t: wired_bus bus;
    - signal u : bit register;

Entity Statements

-- Notes Page --

An entity can contain *passive* statements to perform actions such as timing or validity checks at the interface of a component. Assertion statements in an entity, for example, may be used to check that setup and hold requirements are satisfied.

A passive statement is one which does not change the state of the behavior. Simply put, the execution of a passive statement does not lead to any signal assignments.

SDSP Bus Write Timing

-- Notes Page --

Memory Write:

1. During a Ti state, the CPU places an address on *A_BUS*. T1 is the next state
2. After the leading edge of *phi1*, the CPU asserts *write* to initiate a write activity in the memory. Data to be written is placed on *D_BUS*.
3. T2 states occur until *ready* is asserted by the memory
4. Memory deasserts *ready* on falling edge of *write* and removes address and data from *A_BUS* and *D_BUS* respectively

VHDL Models of the SDSP Microprocessor

IEEE VHDL LRM

VHDL HOME PAGE

- The SDSP Testbench
- The SDSP Behavioral Model
- The SDSP Clock and Memory Models

Two Full Adder Processes

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Defining the *SUM* and *CARRY* functions of a full-bit adder as logic gate circuits can be represented in VHDL as a single sequential assignment statement:

SUM <= *A* xor *B* xor *Cin*;
CARRY <= *A* and *B* or *A* and *Cin* or *B* and *Cin*;

We can represent these two functions in separate process statements (but both in the same architecture, as shown here), or together in the same process statement.

In the example shown, the sensitivity lists are composed of all signals on the right-hand side of the signal assignment statements. We really don't need the explicit sensitivity lists since the default in VHDL is to be sensitive to all right-hand signals. But it doesn't hurt to show them explicitly. In fact, it makes the process easier to understand.

Equivalent Processes

-- Notes Page --

A process with a sensitivity list, as shown in the process on the left, is implemented as a *wait on "sensitivity list"* at the bottom of the process (as shown in the process on the right). This allows every process with a sensitivity list to execute once at the beginning of a simulation and suspend at the bottom waiting for a a relevant signal event to occur. Note that the VHDL standard prohibits the use of both process sensitivity lists and wait statements within the same process.

This should help clarify how a process works. When the simulation begins at time=0, all processes execute until a wait statement is reached. If the wait statement is at the end of the process, the behavior is exactly as if a sensitivity list were used. The wait statement prevents the process from immediately executing from the beginning again. Rather, it waits until one or more of the signals in the wait statement change value.

IEEE VHDL LRM

"wait until" and "wait for"

VHDL HOME PAGE

- What do these do?

```
Summation: PROCESS
 BEGIN
      Sum <= A xor B xor Cin;
      WAIT UNTIL A = '1';
END PROCESS Summation;

Summation: PROCESS
 BEGIN
      Sum <= A xor B xor Cin;
      WAIT FOR 100 ns;
END PROCESS Summation;
```

Bus Resolution

IEEE VHDL LRM

Smoke Generator Fixed

VHDL HOME PAGE

- If a signal has a bus resolution function associated with it, then the signal may have multiple drivers

```
LIBRARY attlib; USE attlib.att_mvl.ALL;
USE WORK.bus_resolution.ALL;

ENTITY bus IS
    PORT (a, b, c : IN MVL; z : OUT MVL);
END bus;

ARCHITETURE fixed OF bus IS
    SIGNAL circuit_node : wired_and MVL;
BEGIN
    circuit_node <= a;
    circuit_node <= b;
    circuit_node <= c;
    z <= circuit_node;
END fixed;
```

Null Transactions

-- Notes Page --

A null transaction is used to deactivate a signal driver. This is analogous to putting a tri-state driver in a high-impedance state. In such a case, the value of the signal is determined by any active drivers. Of course, if there is more than one active driver at any one time, a Bus Resolution Function would be needed.

There are two actions that can take place if all drivers of a signal are disconnected:

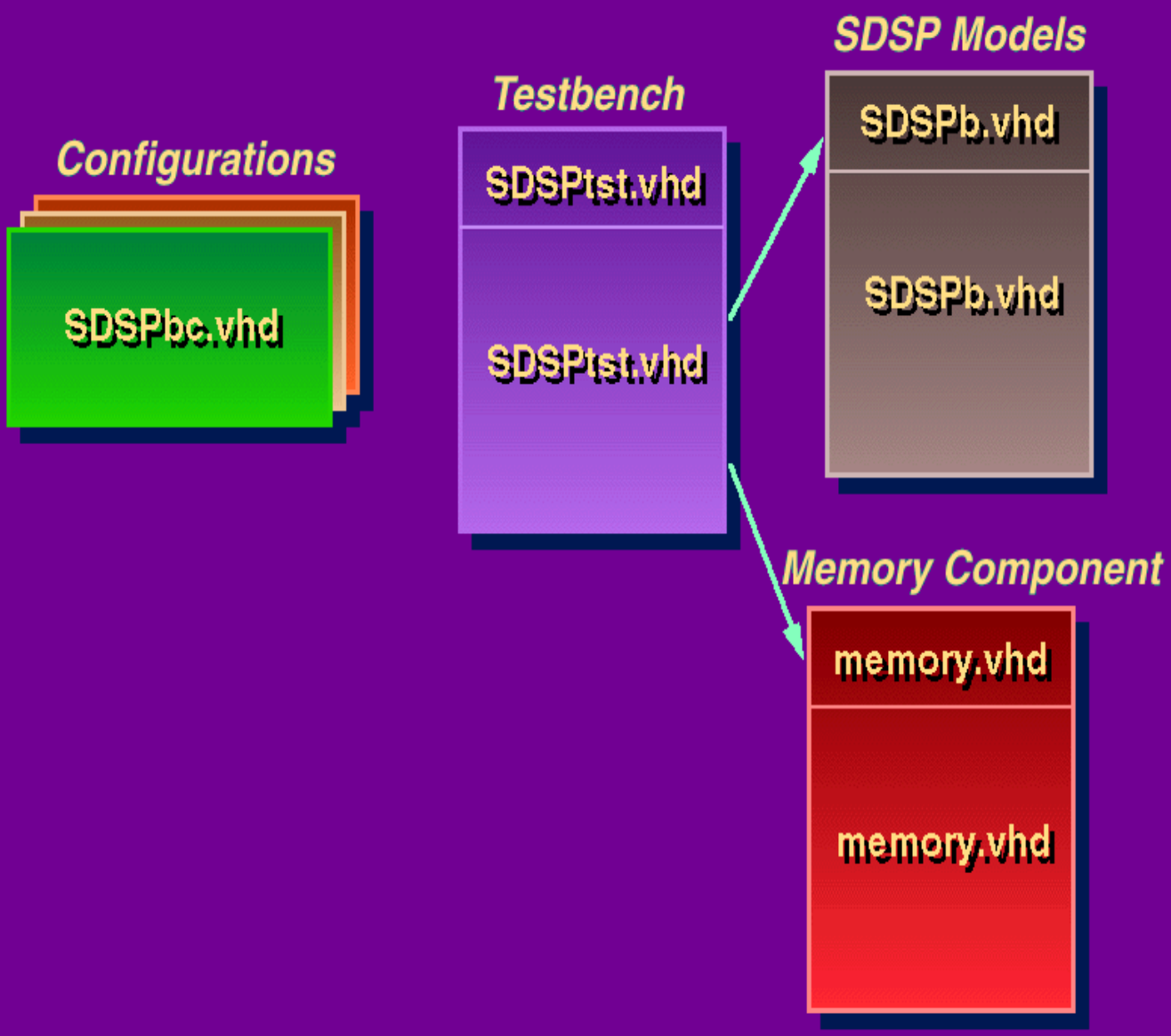
1. Use the last known value
2. Require that a bus resolution function specify a value

The keyword *register* is used if the last known value action is desired, and the keyword *bus* if a bus resolution function must specify a value. These special keywords are used only in the declaration of a signal.

Organization of the SDSP VHDL Model

IEEE VHDL LRM

VHDL HOME PAGE



Exercising the SDSP Model

- Suppose we wish to determine the effect of memory access time on the execution of speed of the SDSP?
  - Modify the SDSP to have a memory access time independent of the SDSP clock
  - Add to the SDSP model a means to determine the number of the clock cycles executed by a DSP benchmark routine
  - Simulate the SDSP using the benchmark for memory access speeds of 20, 40, 60, 80, and 100 nsecs
  - Plot and summarize your results

SDSP Benchmark

-- Notes Page --

The SDSL Benchmark set of slides show a SDSP program, a random number generator, in part to illustrate how to embed timing marks in the benchmark routine such that the VHDL model will report cycle times automatically.

The first thing done in the program is to set R0 to zero followed by a branch to the beginning of the program. *starttrigger* and *donetrigger* are used to notify the SDSP model when the random number generator program is beginning and ending. Our purpose here is to count the total number of clock cycles used by the benchmark program, excluding for the R0 initialization.

A few VHDL statements need to be added to the SDSP memory model to look for specific addresses on the bus. When the *starttrigger* address is seen, a counter is started which increments each clock cycle. When the *stoptrigger* address is seen, the contents of the counter are multiplied by the clock cycle time and reported via an assert or textio write statement.

IEEE VHDL LRM

SDSP Memory Model

VHDL HOME PAGE

```
  USE WORK.SDSP_types.ALL;

   ENTITY memory IS
     GENERIC (Tpd : TIME := unit_delay);
     PORT (d_bus : INOUT bus_bit_32 bus;
           a_bus : IN bit_32;
           read, write : IN BIT;
           ready : OUT BIT);
   END memory:

   ARCHITECTURE behavior OF memory IS
   BEGIN
     PROCESS
       CONSTANT low_address : integer := 0;
       CONSTANT high_address : integer := 65535;
       TYPE memory_array is
         array (INTEGER RANGE low_address TO high_address) of
bit_32;
       VARIABLE mem : memory_array;
       VARIABLE address : INTEGER;
```

```
  BEGIN  -- put d_bus and reply into initial state
        d_bus <= NULL AFTER Tpd;
        READY <= '0' AFTER Tpd;
        WAIT UNTIL (read = '1') OR (write = '1');  -- wait for a
command
        -- dispatch read or write cycle
        address := bits_to_int(a_bus);
        IF address >= low_address AND address <= high_address THEN
          IF write = '1' THEN -- address match for this memory
             ready <= '1' AFTER Tpd;
             WAIT UNTIL write = '0'; -- end of write cycle
             mem(address) := d_bus'delayed(Tpd);  -- sample data
          ELSE  -- read = '1'
             d_bus <= mem(address) AFTER Tpd;   -- fetch data
              ready <= '1' AFTER Tpd;
              WAIT UNTIL read ='0';   -- hold for read cycle
          END IF;
```

```
    END IF;
  END PROCESSS;
END behavior;
```

Exercising the SDSP Model

-- Notes Page --

Now that we have created the SDSP model, we can use it to determine the effect of memory access time on the execution speed of the SDSP, for example.

To do this, we first need to ensure the memory timing can be independently defined.

We then need to add some checks and print statements to determine the number of clock cycles executed by the SDSP when executing a benchmark SDSP program.

We then simulate the SDSP with the object code for the benchmark loaded into the SDSP memory.

Finally, we plot and summarize our results.

IEEE VHDL LRM

"wait until" and "wait for"

-- Notes Page --

VHDL HOME PAGE

Note that the top process suspends indefinitely at the wait statement until there is an event on *A* and *A* has a value of '1';

On the other hand, the bottom process suspends at the wait statement for 100ns (of simulation time, of course) and then proceeds.

Mix and Match

- Within an architecture we have two signals and the following process:

```
DoSomething: PROCESS BEGIN WAIT ON AnotherSignal;
ThisSignal <= '1'; WAIT FOR 10 ns; ThisSignal <=
'0'; WAIT UNTIL (AnotherSignal = '1'); ThisSignal
<= '1'; END PROCESS DoSomething;
```

IEEE VHDL LRM

Bus Resolution Functions

VHDL HOME PAGE

- VHDL uses bus resolution functions to resolve the final value of multiple signal assignments

```
FUNCTION wired_and (drivers : MVL_VECTOR) RETURN MVL IS
    VARIABLE accumulate : MVL := '1';
BEGIN
    FOR i IN drivers'RANGE LOOP
      accumulate := accumulate AND drivers(i);
    END LOOP;
    RETURN accumulate;
End wired_end;
```

- Bus resolution functions may be user defined or called from a package

Organization of the SDSP VHDL Model

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The entire model necessary to simulate the SDSP consists of the testbench (located in file SDSPst.vhd), the SDSP processor model (located in file SDSPb.vhd), and the memory model (in file memory.vhd). A package of types used by all models is not shown here.

IEEE VHDL LRM

The SDSP Testbench

VHDL HOME PAGE

```
USE WORK.SDSP_types.ALL;
  ENTITY SDSP_test IS
  END SDSP_test;

  ARCHITECTURE structure OF SDSP_test IS
     COMPONENT clock_gen
        PORT (phi1, phi2 : OUT BIT; reset : OUT BIT);
     END COMPONENT;
     COMPONENT SDSP
        PORT (d_bus : INOUT bus_bit_32 bus;
              a_bus : OUT bit_32; read, write : OUT BIT;
              fetch : OUT BIT; ready : IN BIT;
              phi1, phi2 : IN BIT; reset : IN BIT);
     END COMPONENT;
     COMPONENT memory
       PORT (d_bus : INOUT bus_bit_32 bus;
             a_bus : IN BIT_32; read, write : IN BIT;
             ready : OUT BIT);
     END COMPONENT;
```

```
SIGNAL d_bus : bus_bit_32 bus;
SIGNAL a_bus : bit_32;
SIGNAL read, write : BIT;
SIGNAL fetch : BIT;
SIGNAL ready : BIT;
SIGNAL phi1, phi2 : BIT ;
SIGNAL reset : BIT;
```

The SDSP Clock Model


```
  USE WORK.SDSP_types.ALL;
   ENTITY clock_gen IS
     GENERIC (Tpw : TIME;       -- clock pulse width
              Tps : TIME);      -- pulse separation between
phases
     PORT (phi1, phi2 : OUT BIT;
              reset : OUT BIT);
   END clock_gen;

  ARCHITECTURE behavior OF clock_gen IS
     CONSTANT clock_period : TIME := 2*(Tpw+Tps);
   BEGIN
     reset_driver : reset <= '1', '0' AFTER 2*clock_period+Tpw;
     clock_driver : PROCESS
     BEGIN
       phi1 <= '1', '0' AFTER Tpw;
       phi2 <= '1', '0' AFTER Tpw+Tps, '0' after Tpw+Tps+Tpw;
       WAIT FOR clock_period;
     END PROCESS clock_driver;
   END behavior;
```

The SDSP Memory Model

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Memory is modeled as an array of 32-bit numbers. The length of the memory is defined as a constant here, but could be made a generic for more modeling flexibility.

As in the processor model, the input-to-output delay of the memory is modeled by *Tpd*. By defining *Tpd* as a generic, one can evaluate SDSP performance where the processor and memory differ in execution speeds.

The memory is usually disconnected from *d_bus* with *ready* deasserted while it waits for a read or write command. Upon receiving a read or write, memory is then read or written.

Mix and Match

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

We show here how wait statements can be used to synchronize the execution of processes, and also how to sensitize a process to signal changes in another process.

In this example, the *DoSomething* process does not execute until *AnotherSignal* changes value. Then we generate a transaction for *ThisSignal* by assigning it a value of '1 and wait for 10 ns. Since the delay time for *ThisSignal <= '1'* is only one delta cycle, *ThisSignal* is updated to have the value '1' on the next delta cycle.

After waiting 10 ns, *DoSomething* assigns a value of '0' to *ThisSignal*; *ThisSignal* will actually take on the new value after one delta cycle. Execution of *DoSomething* is then suspended until *AnotherSignal* becomes '1'. When execution resumes, *ThisSignal* is set to '1' and the process immediately repeats from the top.

IEEE VHDL LRM

Testbench

VHDL HOME PAGE

- Testbenches have three main pusposes
  - Generate stimulus for simulation
  - Apply stimulus to the entity under test
  - Compare output responses with expected values

```
ENTITY testbench IS
-- no PORT statement necessary
END testbench;

ARCHITECTURE example IS testbench
    COMPONENT entity_under_test
      PORT(...)
    END COMPONENT;
BEGIN
    Generate_waveforms_for_test;
    Instantiate_component;
    Monitoring_statements;
END example;
```

Things That Look Alike

- From Project 1's testbench:

```
ARCHITECTURE example
  OF testbench IS
   .
   .
BEGIN
   MakeReset (RESETsignal, 100 ns);

   MakeClock (CLOCKsignal, 10 ns);
   .
   .
   .
END example;
```

```
PROCESS
BEGIN
  MakeReset (RESETsignal, 100 ns);
   WAIT ON RESETsignal;
END PROCESS;

PROCESS
BEGIN
  MakeClock (CLOCKsignal, 10 ns);
   WAIT ON CLOCKsignal;
END PROCESS;
```

Bus Resolution

Smoke Generator

- VHDL does not allow multiple concurent signal assignments to the same signal
  - Multiple sequential signal assignments are allowed

```
LIBRARY attlib; USE attlib.att_mvl.ALL;
-- this code will generate an error
ENTITY bus IS
    PORT (a, b, c : IN MVL; z : OUT MVL);
END bus;

ARCHITECTURE smoke_generator OF bus IS
    SIGNAL circuit_node : MVL;
BEGIN
    circuit_node <= a;
    circuit_node <= b;
    circuit_node <= c;
    z <= circuit_node;
END smoke_generator;
```

IEEE VHDL LRM

Things That Look Alike

-- Notes Page --

VHDL HOME PAGE

Because VHDL is a rich language, there are several ways to say the same thing. This example illustrates how the concurrent VHDL statements shown on the left side (as procedure calls, actually) are equivalent to the one-statement processes shown on the right. Note that the "sensitivity list" for each process (actually shown as a *wait on* statement at the end each process here) includes the signals on the "right hand side" of the concurrent statement (which is actually a procedure parameter in each of the concurrent statements here). That is, the wait statements in the two processes synchronize the process execution with external signal inputs.

Even Signal Assignment Statements!

IEEE VHDL LRM

VHDL HOME PAGE

```
ARFCHITECTURE example OF
    full_adder is

BEGIN

   Summation:  PROCESS( A, B, Cin)
      BEGIN
        Sum <= A xor B xor Cin;
   END PROCESS Summation;

   Carry:  PROCESS( A, B, Cin)
      BEGIN
      Cout <=  (A and B) or
        (A and Cin) or
        (B and Cin);
      END PROCESS Carry;
END example;
```

```
ARCHITECTURE
example OF
full_adder is
BEGIN Sum <= A
xor B xor Cin;
Cout <= (A and
B) or (A and
Cin) or (B and
Cin); END
Example;
```

Map Notes Help

IEEE VHDL LRM

Procedures (cont. 1)

VHDL HOME PAGE

`` ARCHITECTURE behavior OF adder IS BEGIN PROCESS (enable, x, y) BEGIN add_bits3(x, y, enable, result, carry); END PROCESS; END behavior; ```	With parameter passing, it is possible to further simplify the architecture
The parameters must be compatible in terms of data flow and data type	``` PROCEDURE add_bits3 (SIGNAL a, b, en : IN BIT; SIGNAL temp_result, temp_carry : OUT BIT) ```

Even Signal Assignment Statements!

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

As in the previous page, we see that the concurrent signal assignment statements on the right can be described using one-statement processes as seen on the left.

IEEE VHDL LRM

Signal Assignment Statements

VHDL HOME PAGE

```
ARCHITECTURE stuff OF my_entity IS
    SIGNAL ThisBit : BIT;
    SIGNAL ThisBitVector : BIT_VECTOR(1 to 5)
    SIGNAL ThisInteger : INTEGER;
    SIGNAL ThisString : STRING(1 to 4);
BEGIN
   .
   .
END stuff;
```

```
ThisBit <= '1';

ThisBitVector <= "10010";


ThisInteger <= 567 after 10ns;

ThisString <= "VHDL" after 10ns,
                      "is  " after 20ns,
                      "fun!" after 30ns;
```

Signal Assignment Statements

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The structure of signal assignment statement allows some flexibility. However, the signal type of the result on the right hand side must match the type of the signal being assigned. This is illustrated in the first two assignments shown on the right.

The third assignment shows the use of a single *after* clause used to control how much simulation time must pass before the assigned signal takes on its new value.

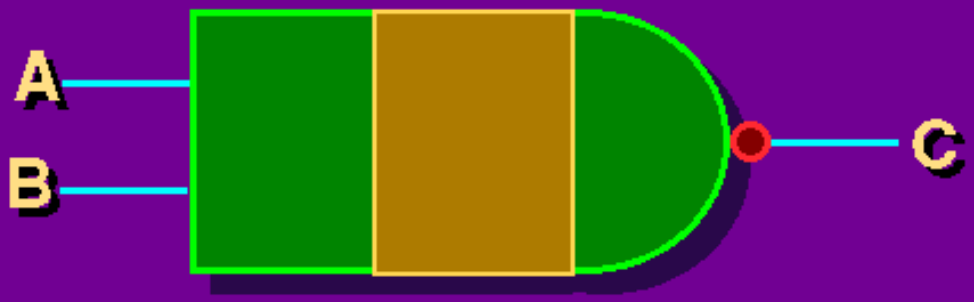
As seen in the fourth example, multiple assignments can be made in a single statements by separating them with commas. This sequence of assignments is called a "waveform".

If a signal assignment statement has no *after* clause, a clause equivalent to "after 1 delta cycle" is implied. Delta cycles are key to the VHDL timing model and have been previously discussed in the VHDL Basics module.

IEEE VHDL LRM

Inertial vs Transport Delays

VHDL HOME PAGE



Inertial Timing

```
ENTITY nand2 IS
    PORT(A, B:IN bit; C:OUT bit);
END nand2;

ARCHITECTURE behaviour OF nand2 IS
BEGIN
    C <= not(A and B) AFTER 25ns;
END behaviour;
```

Transport Timing

```
ENTITY nand2 IS
    PORT(A, B:IN bit; C:OUT bit);
END nand2;

ARCHITECTURE behaviour OF nand2 IS
BEGIN
    C <= TRANSPORT not(A and B) AFTER 25 ns;
END behaviour;
```


Map
Notes
Help

Inertial vs Transport Delays

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

VHDL can model two types of delay in a component:

- Inertial delay - if two events occur on an input of the component with an interval time less that the defined delay, the output will not reflect either input event, and
- Transport delay - Any event on an input of the component will be reflected on the output.

The default timing type is inertial. If you want to model a transport delay, use the keyword "transport".

Transport delays are typically used to synchronize timing between VHDL processes and to model systems at high levels where inertial delay effects are ignored.

IEEE VHDL LRM

Subprograms

VHDL HOME PAGE

- Similar to subprograms found in other languages
- Allow repeatedly used code to be referenced many times without duplication
- Break down large chunks of code in small, more manageable parts
- VHDL provides functions and procedures for use

IEEE VHDL LRM

Functions

VHDL HOME PAGE

- Produce a single return value
- Called by expression
- Can not modify the parameters passed to it
- Requires a RETURN statement

```
FUNCTION add_bits (a, b : IN BIT) RETURN BIT IS
BEGIN -- functions can NOT return multiple values
        RETURN (a XOR b);
End add_bits;
```

```
FUNCTION add_bits2 (a, b : IN BIT) RETURN BIT IS
  VARIABLE result : BIT; -- variable is local to function
    BEGIN
      result := (a XOR b);
      RETURN result; -- the two functions are equivalent
END add_bits2;
```

IEEE VHDL LRM

Functions (cont. 1)

VHDL HOME PAGE

```
ARCHITECTURE behavior OF adder IS
BEGIN
    PROCESS (enable, x, y)
    BEGIN
    IF (enable = '1') THEN
      result <= add_bits (x, y);
      carry <= x AND y;
    ELSE
      carry, result <= '0';
    END PROCESS;
END behavior;
```

```
FUNCTION add_bits

(a, b : IN BIT)
```

- Functions must be called by other statements
- Parameters use positional association

IEEE VHDL LRM

Procedures

VHDL HOME PAGE

- Produce many output values
- Are invoked by statements
- May modify the parameters

```
PROCEDURE add_bits3 (SIGNAL a, b, en : IN BIT;
    SIGNAL temp_result, temp_carry : OUT BIT) IS

BEGIN -- procedures can return multiple values
    temp_result <= (a XOR b) AND en;
    temp_carry <= a AND b AND en;

END add_bits3;
```

- Do not require a RETURN statement

The SDSP Testbench

-- Notes Page --

The testbench declares the three components - *clock*, *SDSP* processor, and *memory*. The user-defined types shown here to declare the signal objects are defined in the package SDSP_types.

IEEE VHDL LRM

Testbench Body

VHDL HOME PAGE

```
 BEGIN
    cg : clock_gen
       PORT MAP (phi1 => phi, phi2 => phi2, reset => reset);
    proc : SDSP
       PORT MAP (d_bus => d_bus, a_bus => a_bus,
                 read => read, write => write, fetch => fetch,
ready => ready,
                 phi1 => phi1, phi2 => phi2, reset => reset);
   mem : memory
       PORT MAP (d_bus => d_bus, a_bus => a_bus,
                 read => read, write => write, ready => ready);
 END structure;
```

Testbench Body

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The body of the testbench is simply the instantiation of the three system components. It should be noted that there is no mechanism defined for generating test vectors or reading them from a file. To simulate the SDSP, the user must first write the SDSP object code instructions into simulated memory using the VHDL simulator support system, and then start the simulation.

IEEE VHDL LRM

The SDSP Behavioral Model

VHDL HOME PAGE

```
USE WORK.SDSP_types.ALL;
 ENTITY SDSP IS
   GENERIC (Tpd : Time
                := unit_delay);
   PORT ( d_bus : inout
             bus_bit_32 bus;
    a_bus : out bit_32;
    read, write : out bit;
    fetch : out bit;
    ready : in bit;
    phi1, phi2 : in bit;
    reset:in bit);
 END SDSP;
```

```
USE WORK.SDSP_types.ALL;
 ARCHITECTURE behavior OF SDSP IS
   SUBTYPE reg_addr is NATURAL RANGE 0 to 255;
   TYPE reg_array IS array (reg_addr) OF bit_32;
 BEGIN
   PROCESS
     VARIABLE reg : reg_array;
     VARIABLE PC : bit_32;
     VARIABLE current_instr : bit_32;
     VARIABLE op: bit_8;
     VARIABLE r3, r1, r2 : reg_addr;
     VARIABLE i8 : integer;
     ALIAS cm_i : BIT IS current_instr(19);
     ALIAS cm_V : BIT IS current_instr(18);
     ALIAS cm_N : BIT IS current_instr(17);
     ALIAS cm_Z : BIT IS current_instr(16);
     VARIABLE cc_V, cc_N, cc_Z : bit;
     VARIABLE temp_V, temp_N, temp_Z : bit;
     VARIABLE displacement : bit_32;
     VARIABLE effective_addr : bit_32;
```

The SDSP Behavioral Model

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The next few slides show the behavioral model for the SDSP. The entity description only provides a description of the ports and the single generic, *Tpd*, which is the delay time in the processor between input events and output signal changes.

The address bus can be modeled as a simple bit vector, but the data bus must be defined as a resolved bit vector since it can be driven either by the processor or the memory. The bus indication for *d_bus* in the port declaration means that all bits of *d_bus* can be disconnected at the same time. Note that aliases are used to provide meaningful names for the fields of the instruction.

The next few slides will show the memory *read* and *write* procedures, a representative data-path procedure, *add*, and finally the main routine in the model which executes the fetch-decode-execute cycle of the SDSP.

The SDSP Read Memory Procedure

`PROCEDURE memory_read (addr : IN bit_32;` `fetch_cycle : IN BOOLEAN;` `result : OUT bit_32) IS` `BEGIN`	
`a_bus <= addr AFTER Tpd; --start bus cycle` `fetch <= bool_to_bit(fetch_cycle) AFTER Tpd;` `WAIT UNTIL phi1 = '1';`	Place address on a_bus
`IF reset = '1' THEN RETURN; END IF;` `read <= '1' after Tpd; -- T1 phase` `WAIT UNTIL phi1 = '1';` `IF reset = '1' then RETURN; END IF;`	Assert read signal
`LOOP -- T2 phase` `WAIT UNTIL phi2 = '0'; -- end of T2` `IF reset = '1' then RETURN; END IF;` `IF ready = '1' then result := d_bus; EXIT;` `END IF;` `END LOOP;` `WAIT UNTIL phi1 = '1';` `IF reset = '1' THEN RETURN; END IF;`	Place data on result bus when ready
`read <= '0' AFTER Tpd; -- Ti phase at end of cycle` `END memory_read;`	Deassert read signal

The SDSP Read Memory Procedure

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The memory read and write procedures exactly implement the timing requirements presented several slides back. Since asynchronous events cannot be modeled as such in VHDL processes or procedures, we emulate an asynchronous reset by checking for it after each micro-operation.

It is instructive to examine the flow of execution here with the timing model diagram to observe how direct the behavior mapping can be.

SDSP Write Memory Procedure

IEEE VHDL LRM

VHDL HOME PAGE

`PROCEDURE memory_write (addr : IN bit_32; data : IN bit_32) IS`	
`BEGIN` `  a_bus <= addr AFTER Tpd;  -- start bus cycle` `  fetch <= '0' AFTER Tpd;` `  WAIT UNTIL phi1 = '1';`	Place address on a _bus
`  IF reset = '1' THEN RETURN; END IF;` `  write <= '1' AFTER Tpd;  -- T1 phase` `  WAIT UNTIL phi2 = '1';` `  d_bus <= data AFTER Tpd;` `  WAIT UNTIL phi1 = '1';`	Assert write signal and place data on d_bus
`  IF reset = '1' THEN RETURN; END IF;` `  LOOP -- T2 phase` `  WAIT UNTIL phi2 = '0';` `  IF reset = '1' THEN RETURN; END IF;` `  EXIT WHEN ready = '1';  -- end of T2` `  END LOOP;` `  WAIT UNTIL phi1 = '1';`	Wait until data is written and ready signal is asserted
`  IF reset = '1' THEN RETURN; END IF;` `  write <='0' AFTER Tpd; -- Ti phase at end of cycle` `  d_bus <= null AFTER Tpd;` `END memory_write;`	Deassert write signal and detach d_bus

The SDSP Write Memory Procedure

-- Notes Page --

Writing memory is coded similarly to the read memory procedure. The null assignment statement at the end of the procedure disconnects the processor from *d_bus* after the write cycle is complete. In other words, the processor ceases to drive the *d_bus* port.

SDSP Add Procedure

```
  PROCEDURE add (result : INOUT bit_32;
                 op1, op2 : IN INTEGER;
                 V, N, Z : OUT BIT) IS
   BEGIN
     IF op2 > 0 and op1 > integer'high-op2 THEN   -- positive
overflow
       int_to_bits(((integer'low+op1)+op2)-integer'high-1,
result);
       V := '1';
     ELSIF op2 < 0 and op1 < integer'low-op2 THEN -- negative
overflow
       int_to_bits(((integer'high+op1)+op2)-integer'low+1,
result);
       V := '1';
     ELSE
       int_to_bits(op1 + op2, result);
       V := '0';
     END IF;
     N := result(31);
     Z := bool_to_bit(result = X"0000_0000");
   END add;
```

SDSP Add Procedure

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The *add* procedure is shown here as a representative of the set of procedures necessary to implement the behavior of each op code.

Data is represented in 2's-complement form. Thus, most of the VHDL code is used to check for various boundary conditions resulting from the operation.

In this procedure, the first section of the IF-THEN-ELSE statement checks for postive overflow, the second for negative overflow and the last performs addition without overflow of either kind.

If there is positive overflow then it requires that OP2 be greater than zero and OP1 be greater than the difference between the highest possible integer represented with 32 bits and OP2. This is the only possible instance where positive overflow can occur. If positive overflow does occur, then the result must adjusted to reflect its final representation in 2's complement form. This modification is shown as the first argument to the int_to_bits routine and is left as an exercise for the user to verify its correctness. In this case the *V* bit is set to '1' to notify the user that overflow has occurred.

If there is negative overflow then it requires that OP2 be less than zero and OP1 be less than the difference between the lowest possible integer represented with 32 bits and OP2. It again is an exercise for the user to verify the correctness of the first argument to the int_to_bits routine. In this case the *V* bit is set to '1' to notify the user that overflow has

occurred..

If no overflow will result from the sum, the operands are added and the overflow bit, *V*, is set to '0'. Finally, the *N* and *Z* bits are appropriately set and the procedure exits.

SDSP Behavioral Model


```
  BEGIN
     -- check for reset active
     IF reset = '1' THEN
       read <= '0' AFTER Tpd;
       write <= '0' AFTER Tpd;
       fetch <= '0' AFTER Tpd;
       d_bus <= null AFTER Tpd;
       PC := X"0000_0000";
       WAIT UNTIL reset = '0';
     END IF;
     -- fetch next instruction
     memory_read(PC, true, current_instr);
     IF reset/= '1' THEN
        add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
       --decode & execute
       op := current_instr(31 DOWNTO 24);
       r3 := bits_to_natural(current_instr(23 DOWNTO 16));
       r1 := bits_to_natural(current_instr(15 DOWNTO 8));
       r2 := bits_to_natural(current_instr(7 DOWNTO 0));
       i8 := bits_to_int(current_instr(7 DOWNTO 0));
```

```
  CASE op IS
     WHEN op_add =>
       add(reg(r3), bits_to_int(reg(r1)), buts_to_int(reg(r2)),
cc_V, cc_N, cc_Z);
     WHEN op_addq =>
       add(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
     WHEN op_sub =>
       subtract(reg(r3), bits_to_int(reg(r1)),
bits_to_int(reg(r2)), cc_V, cc_N, cc_Z);
               . . . . . .
     WHEN op_land =>
       reg(r3) := reg(r2);
       cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
               . . . . . .
     WHEN op_ld =>
       memory_read(PC, true, displacement);
       IF reset /= '1' then add (PC, bits_to_int(PC), 1, temp_V,
temp_N, temp_Z);
```

```
       add(effective_addr, bits_to_int(reg(r1)),
       bits_to_int(displacement), temp_V, temp_N, temp_Z);
               . . . . . .
```

```
        .......
      WHEN op_bi => memory_read(PC, true, displacement);
          IF reset /= '1' THEN
          add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
         add(effective_addr, bits_to_int(reg(r1)),
bits_to_int(displacement),
                        temp_V, temp_N, temp_Z);
          IF ((cm_V and cc_V) OR (cm_N and cc_N) OR (cm_Z and
cc_Z)) = cm_i
   THEN
             PC := effective_addr;
           END IF;
         END IF;
       WHEN op_brq =>
       add(effective_addr, bits_to_int(PC, i8, temp_V, temp_n,
temp_Z);
        if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
= cm_i
   THEN
          PC := effective_adr;
       END IF;
         .........
     WHEN OTHERS => ASSERT false REPORT "illegal instruction"
SEVERITY WARNING;
     end case;
     end if; -- reset /= '1'
     end process;
   end behavior;
```

SDSP Behavioral Model

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Just as with the read and write memory procedures, we check for reset whenever a wait condition exists in the process.

The process executes a typical processor cycle -- fetch next instruction, decode the instruction, and execute the instruction. In this model, decoding is performed by doing type conversion on slices of the instruction, and execution is performed using a VHDL case statement.

Not all of the instructions are shown since there are quite a few of them. But the general pattern can be observed.

For example, the ADD operation is performed simply by making a call to the *add* procedure. Note that heavy use of casting is done to translate variable types passing in and out of procedures.

The *load* operation is shown here to demonstrate the activities that must take place. Two memory reads are performed; the first is to obtain the displacement, and the second is to fetch the desired data.

A branch instruction operates in several steps.

1. Read the displacement from memory
2. Calculate the condition
3. Set the PC accordingly

Note the use of the *others* clause at the end of the case statement. Even if all op-code patterns were used in the case statement (which is not the case here), it is good modeling practice to always use an *others* clause to report anomalous modeling behavior.

The SDSP Clock Model

-- Notes Page --

Note that the clock period is defined as twice the clock pulse width, *Tpw*, and a very small separation between clock pulses (to ensure non-overlapping clock phases), *Tps*. Since these values are constant, we define them as generics.

The clock is generated by setting *phi1* and *phi2* to '1' or '0' and using after clauses to shift the phase of *phi2* relative to *phi1*. The process executes once each clock period to create the next clock cycle.

System Level VHDL - Module 4

IEEE VHDL LRM

VHDL HOME PAGE

Table of Contents

IEEE VHDL LRM

VHDL HOME PAGE

System Level VHDL - Module 4

This module was prepared as part of the RASSP Education & Facilitation effort.

Version 1.0

Toolbar Functionality

	Takes the user up one hierarchical level in the presentation.
	Takes the user to the previous section of the presentation.
	Takes the user to the previous slide in the presentation.
	Takes the user to a listing of all slides with links to each slide.
	Takes the user to a visual representation of the organization of the slide presentation.
	Takes the user to a document, further explaining the information contained within the slide.
	Brings the user to this document, containing information on the use of the toolbar.
	Takes the user to the next slide in the presentation.
	Takes the user to the next section of the presentation.
	Takes the user down one hierarchical level in the presentation.

System Level VHDL - Module 4

Outline

What do We Mean by "Systems Modeling?"

IEEE VHDL LRM

VHDL HOME PAGE

- A system is a group of interdependent items which together carry out a set of functions over time to produce desired results given a set of inputs.
- A DSP system is a system whose function is to process digital signals
- For RASSP, we require the description of a system to include the behavior of the system in addition to the structural or physical description
  - Algorithmic, or
  - Uninterpreted (e.g., modeled by a queuing network or Petri net), or
  - Interpreted as functions

System Level VHDL - Module 4

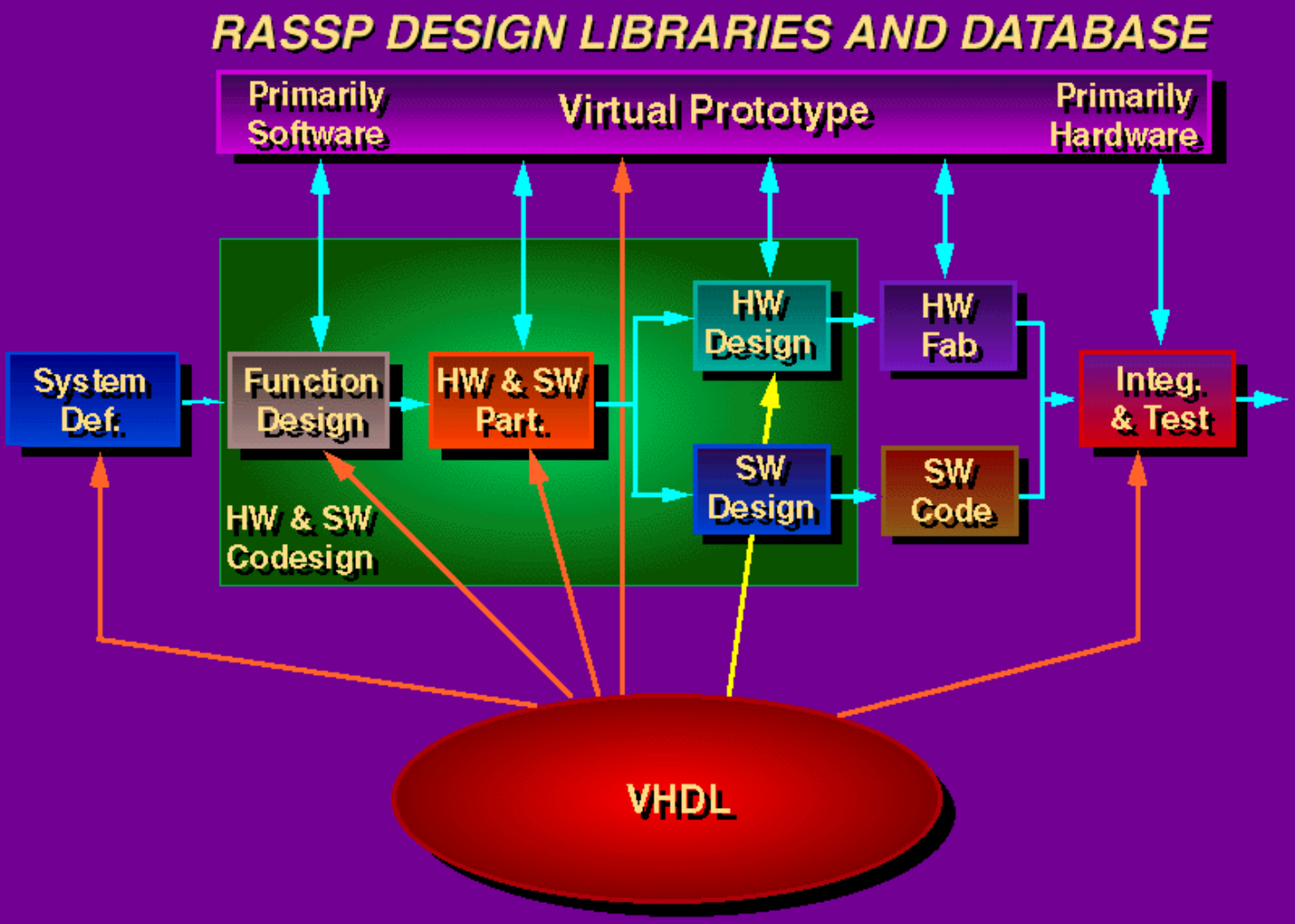
Module Goals

- Understanding of the various system oriented VHDL constructs
- Comprehension of the advantages of using VHDL at the system level
- Familiarity with how VHDL is being used at the system level

IEEE VHDL LRM

RASSP Roadmap

VHDL HOME PAGE



Module Goals

-- Notes Page --

This is the fourth in the series of VHDL instructional modules prepared by the Rapid Prototyping of Application Specific Signal Processors (RASSP) Education & Facilitation team. Building on the previous three modules, this module presents examples in which VHDL is used to model the system at high levels of abstraction.

What do we mean by "Systems Modeling"?

-- Notes Page --

This slide defines how the term "system" is used in this module. The RASSP program is concerned primarily with signal processors. Digital signal processors (DSPs) are special-purpose processing units specifically designed to run signal processing algorithms efficiently (e.g. Fast Fourier transforms, discrete cosine transforms, etc.).

Describing RASSP Systems

- VHDL is a useful hardware description language for the description and simulation of RASSP systems at the system level
- There are other means of describing and simulating at the system level. For example:
  - Verilog - an HDL based on C
  - PGM - a graphical queuing network language
  - VSPEC - a formal language for specifying the constraints and input/output relations of a system
  - Ada - a software programming language rich enough to describe DSP products at the system level
  - ADEPT - An uninterpreted text and graphics language based on VHDL and translatable to VHDL
  - PERFSIM - a queuing network language translatable to VHDL

Describing RASSP Systems

-- Notes Page --

Although VHDL is a versatile hardware description language and is well suited to describing systems at high levels of abstraction, several alternatives exist for the system modeler. Many of the alternatives listed in this slide, however, are not well suited for incrementally refining a system description as detail is added to the design.

Advantages of using VHDL

- VHDL offers several advantages to the system level designer
  - Standard language
  - Fully expressive language
  - Hierarchical
  - Configurable
  - Tool availability
  - Consistency and completeness checks automatic
  - Tight coupling to lower levels of design
  - Supports hybrid modeling

Styles and Approaches

- Model - representation of an item in some form other than the form in which it is to ultimately exist
  - set of equations
  - computer program
  - schematic diagram
  - queuing network
- Model classification
  - Interpreted - the model defines the value of system variables for all time
  - Uninterpreted - the system variables are undefined in the model over some interval(s) of time
- What's the purpose of interpreted models?
  - They very concisely represent the system
  - They simulate *much* faster
  - They provide a mechanism for system conceptualization

Types of System Models

- Formal Models
  - Mathematical formulation
  - Defines constraints on function and performance
  - Defines relations of outputs to inputs
  - Has been largely of academic interests; emerging in industrial practice
- Uninterpreted Models
  - Behavior described as tokens flowing through a queuing network or Petri network
  - Input/output relationships are largely ignored
  - Primary interest is estimating timing delays between inputs and outputs in the system being described
  - Has been largely of academic interest; emerging in industrial practice
- Interpreted models
  - Actual behavior is described, at some level of abstraction
  - Primary interest is defining functions between inputs and outputs
  - Has been standard modeling approach in industry for 20+ years

IEEE VHDL LRM

Styles and Approaches

-- Notes Page --

VHDL HOME PAGE

To provide a consistent use of terms within this module, some definitions are provided in this slide.

Uninterpreted models represent the flow of information in a system without regard to the content of the information. Familiar examples of these types of models are queuing networks and Petri nets.

Models

- Uninterpreted models

  An uninterpreted model is one that does not model actual data values or data-related functionality either internally or externally. Only control information and control functionality are modeled.

  - Generally used to study information flow and performance (and also known as performance models)
  - Based on Petri net or queuing net theory

- Interpreted models

  An interpreted model is one that models actual data values and data-related functionality and control-related functionality both internally and externally.

  - Used to study function, timing, and performance from high to low abstraction levels

[Hein95]

Reprinted with permission.

System Level VHDL Constructs

- For all examples in this section, assume the existence of a multi-values logic package ATT_MVL type MVL is 'U', '0', '1', 'Z');
  - type MVL_VECTOR is array (NATURAL range <>) of MVL;
  - overloaded logical operators for both MVL and MVL_VECTOR
  - overloaded "+" and "-" arithmetic operators on MVL_VECTOR
  - integer-to-MVL_VECTOR conversion functions (int2MVL, MVL2int)

Types of System Models

-- Notes Page --

Some of the many forms in which systems can be described are shown in this slide. Many of these are easily implemented in VHDL.

Models

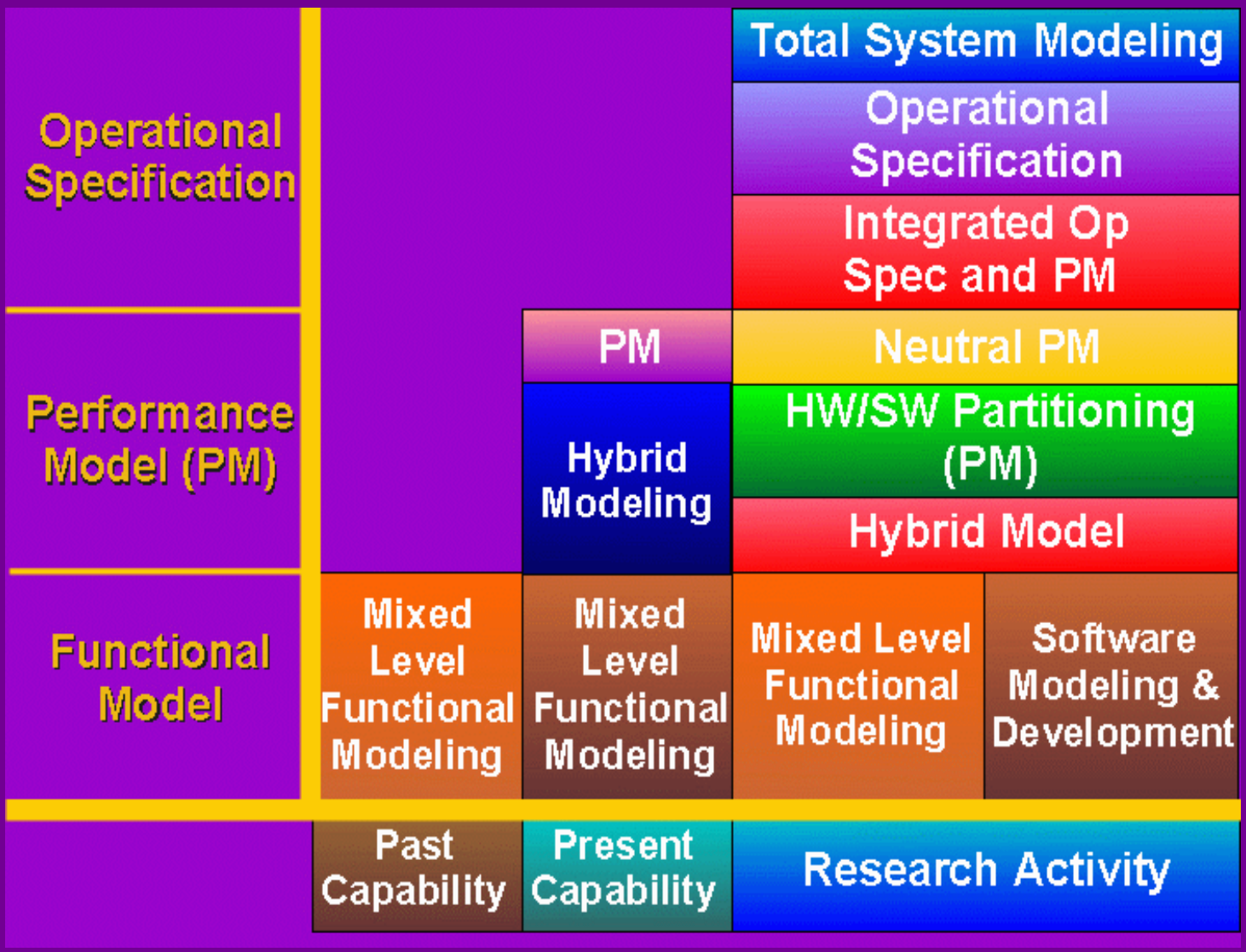
-- Notes Page --

This slide further contrasts uninterpreted and interpreted models.

IEEE VHDL LRM

Evolution of Approaches

VHDL HOME PAGE



System Level VHDL Constructs

-- Notes Page --

The ATT_MVL logic library supports high-level system modeling in VHDL and will be the source of some of the syntax examples in the next few slides. This logic package has multiple logic values and various functions defined to convert the ATT package to integers. Also, various operators are overloaded to handle the ATT package.

[Bhasker95]

IEEE VHDL LRM

Aliases

VHDL HOME PAGE

- Aliases can significantly improve the readability of VHDL descriptions by using a shorthand notation for names
- Aliases allow reference to named items in different ways:

```
SIGNAL data_word: mvl_vector(15 DOWNTO 0);
ALIAS data_bus: mvl_vector (7 DOWNTO 0) IS data_word (15 DOWNTO
8);
```

- Aliases can rename any named item except labels, loop parameters, and generate parameters

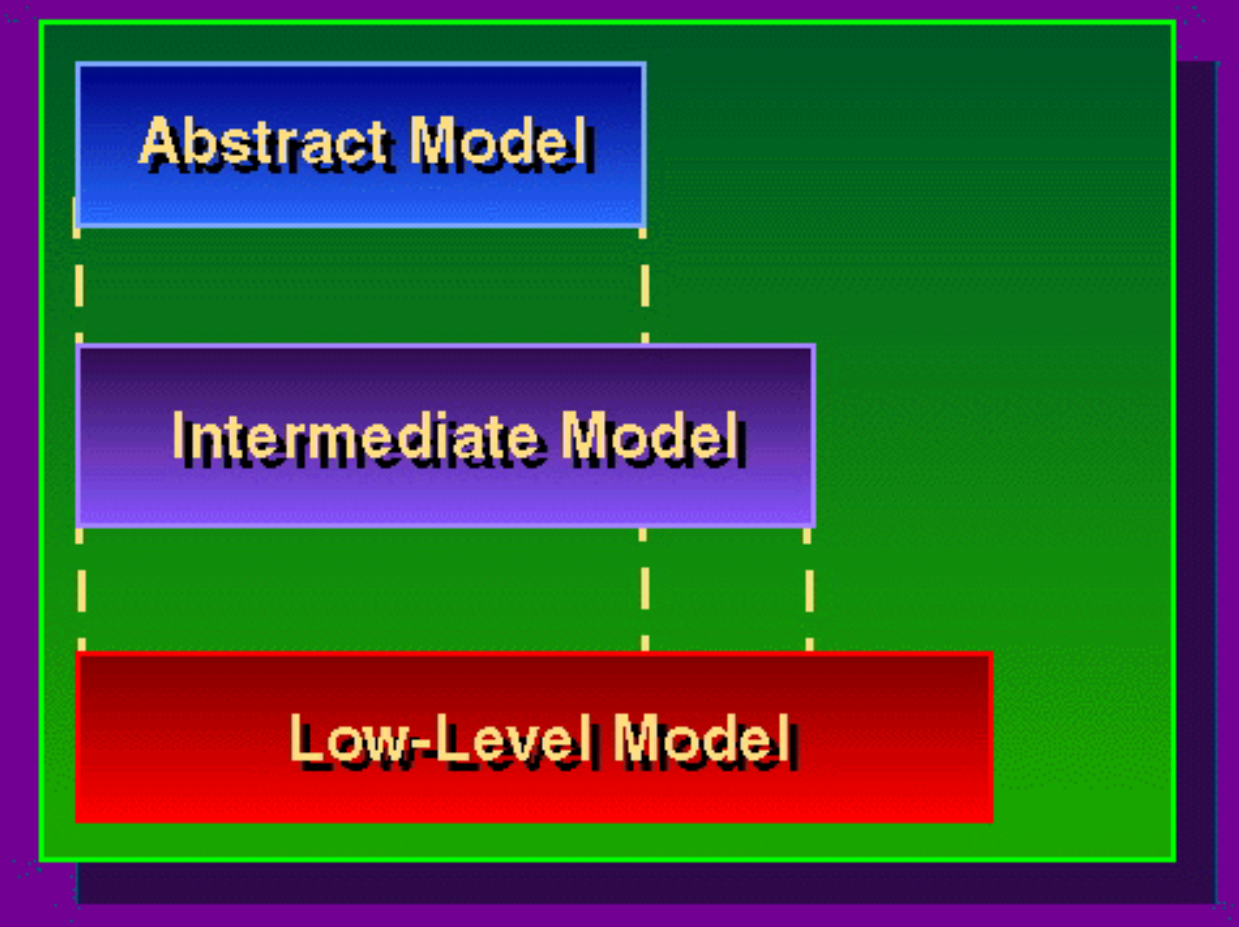
IEEE VHDL LRM

Advantages of Object Oriented VHDL

VHDL HOME PAGE

- Benefit: Better for Incremental Development
- Enables: Abstract modeling & rapid prototyping

Reuse behavior defined in abstract models.
Maintain conceptual relation between models.
Recoding behavior wastes time and introduces errors.




Map
Notes
Help

Summary

- VHDL is suitable for modeling at the system level due to its acceptance as standard and the number of design tools available
- Records provide a concise way for tokens to be implemented in uninterpreted modeling
- Bus resolution functions allow for user defined bus arbitration
- Shared variables are a new VHDL construct specifically targeted at system level modeling

Applications of System Level VHDL

IEEE VHDL LRM

VHDL HOME PAGE

- UVA ADEPT
- Honeywell PML

Converting OO-VHDL to to Synthesizable VHDL

- Remove any reliance on the message queue
  - *Immediate* messages, *select* and *accept* statements
- Remove any hierarchy-spanning messages
  - A component should only communicate with parent, child, or sibling
- Convert the messaging protocol to a signaling protocol
- Convert operations to subprograms
- Convert instance variables to signals or variables, as appropriate
- Add a clock

Applications of System Level VHDL

-- Notes Page --

Two performance-oriented VHDL-based system level description methodologies will be presented in this section as additional examples of the utility of the abstractions supported by VHDL.

UVA ADEPT

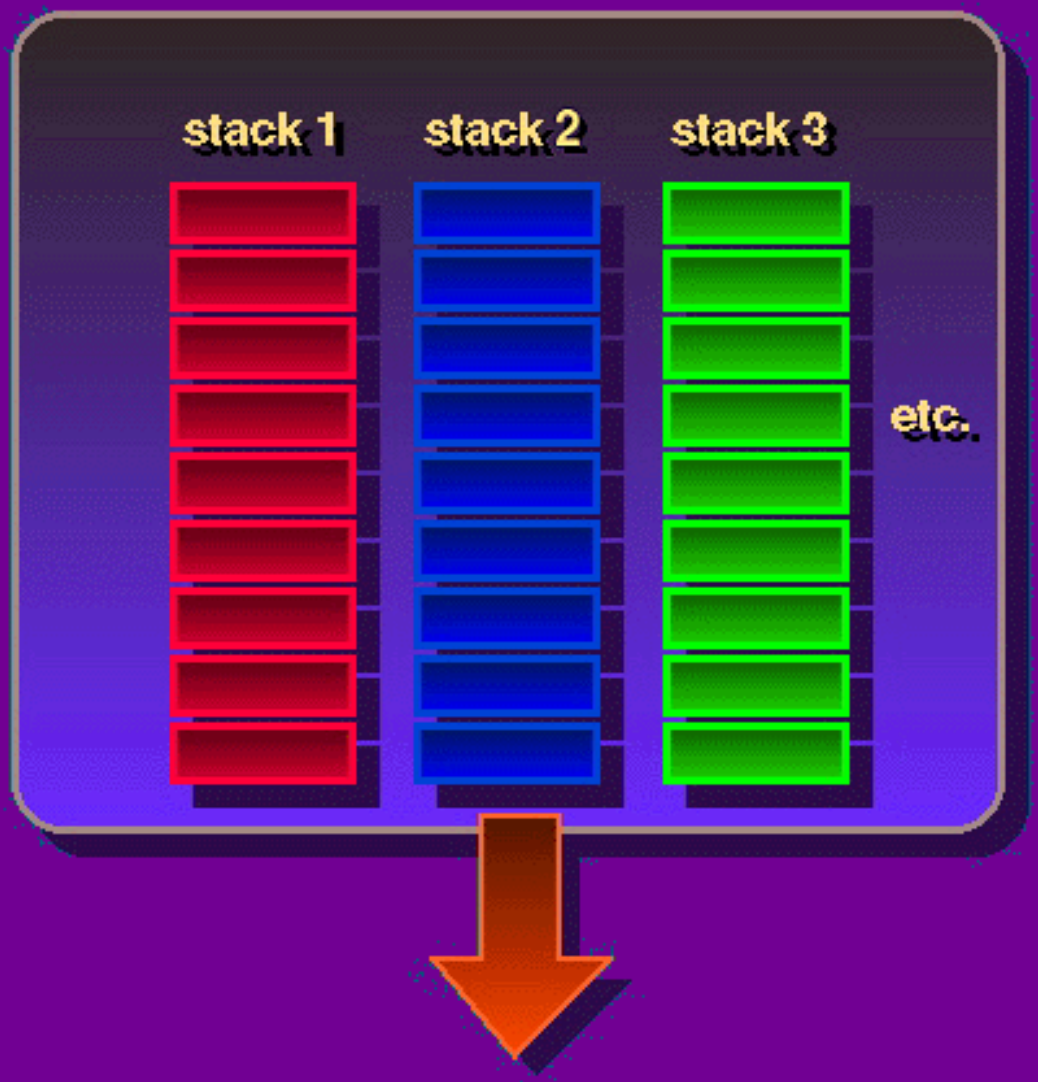
Primitive Modules

- ADEPT has several types of primitives:
  - Control
  - Color
  - Delay
  - Fault
  - Hybrid
  - Miscellaneous
- The ADEPT primitives all have an equivalent Petri Net and VHDL description
- ADEPT hides the details of the underlying Petri Net and VHDL

Honeywell PML
More Functional Memory

IEEE VHDL LRM

VHDL HOME PAGE



Honeywell

```
PACKAGE DataIO_defs IS
  -- deferred constant
  CONSTANT IPC_BUFFER SIZE : INTEGER;
  SUBTYPE memory_identifier_type IS INTEGER;
END PACKAGE DataIO_defs;

PACKAGE BODY DataIO_defs IS
  CONSTANT IPC_BUFFER_SIZE : INTEGER := 4096;
END PACKAGE BODY DataIO_defs;
```

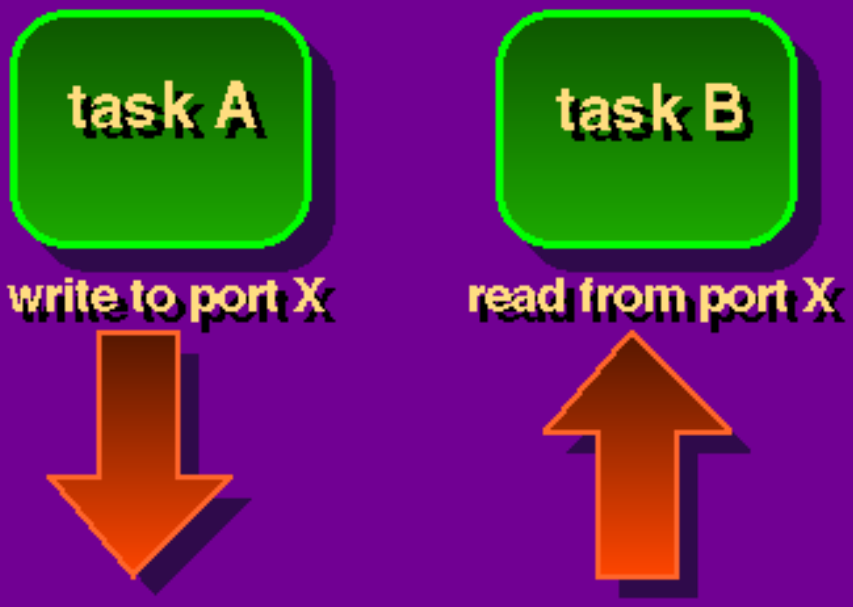
defined in dataio_defs

- The application can partition the Functional Memory into as many stacks as needed. Each stack gets its own unique ID number
- Upon demand (after a write command for a new stack ID), the Functional Memory will dynamically allocate a block of memory
  - The size depends on the IPC_BUFFER_SIZE deferred constant from the dataio_defs package
  - Stack sizes only limited by physical memory
- No concurrent accesses to stacks with identical IDs

Honeywell PML Functional Memory Operations

IEEE VHDL LRM

VHDL HOME PAGE




```
PROCEDURE
write (SIGNAL ioport: INOUT IPC_integer_type;
              item: IN integer_vector;
              ID: IN memory_identifier_type:= 0;
              offset: IN natural := 0;
              count: IN natural := 0);

PROCEDURE
read (SIGNAL ioport: INOUT IPC_integer_type;
              item: IN integer_vector;
```

```
ID: IN memory_identifier_type := 0;
offset: IN natural:= 0;
count: IN natural := 0);
```

defined in dataio packages

- Two simple procedures: read and write
- Simple parameters:
  - ioport, the signal to the Functional Memory
  - item, the vector (integer or real) of data to write/read
  - ID, indicating which memory stack to access
  - offset, which will offset the first memory index by the number given
  - count, indicates the number of vector elements to read/write. If count=0, then by default the operation will use the vector size as the count number
  - Only ioport and item are required parameters

Honeywell PML:

More Functional Memory

-- Notes Page --

In using Functional Memory, space can be allocated dynamically as needed, in blocks of a predefined IPC_BUFFER_SIZE.

[Honeywell95]

Summary

-- Notes Page --

This instructional module has illustrated the versatility of VHDL in supporting abstraction and information encapsulation to facilitate the description of complex systems. Example system design and description methodologies based on VHDL were included primarily to illustrate the VHDL constructs used to support system level modeling.

IEEE VHDL LRM

References

VHDL HOME PAGE

References:

[Berge93] J-M., Fonkoua, A., Maginot, S., Rouillard, J., *VHDL '92: The New Features of the VHDL Hardware Description Language*, Kluwer Academic Publishers, 1993.

[Bhasker95] Bhasker, J., *A VHDL Primer*, Prentice Hall, 1995.

[Hein95] Hein, Karl - Lockheed-Martin ATL, Carpenter, Todd - Honeywell Technology Center, Kalutkiewicz - Lockheed Sanders, Madisetti, Vijay - Georgia Technology Institute, •"RASSP VHDL Modeling Terminology and Taxonomy-Revision 1.0", *Proceedings of the 2nd Annual RASSP Conference*, July 24-27, 1995. A more updated version of this document was published in the *Proceedings of the VIUF Fall 1996 Conference* and was titled, "Effecting VHDL Model Interoperability in RASSP through a Common Modeling Taxonomy". This document is also available on-line at http://rassp.scra.org/public/atl/taxonomy.html.

[Honeywell95] *Honeywell Performance Modeling Library*, 1995.

[Navabi93] Navabi, Z., *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill, 1993.

[UM93] Cutright, E.D., Rao, R., Johnson, B.W., Aylor, J.H., *A Handbook on the Unified Modeling Methodology Building Block Set*, CSIS, University of Virginia, 1993.

For further reading:

Carpenter, T., Rose, F., Steeves, T., *Performance Modeling with VHDL*, Honeywell Systems and Research Center, 1994.

IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993.


Map
Help

Advantages of Using VHDL

-- Notes Page --

Some of the advantages in using VHDL as a description language include its versatility and the fact that it is an accepted standard with broad support from both government and industry.

Evolution of Approaches

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

In this slide the vertical axis consists of three modeling categorizations, Functional Model, Performance Model, and Operational Specification. The horizontal axis indicates the availabilities of various tools and methodologies supporting the categorizations on the vertical axis. Note that the trend is towards additional support of system modeling at more abstract levels.

Aliases

-- Notes Page --

VHDL provides the *alias* construct to enhance readability in VHDL descriptions. VHDL supports two types of *aliases* listed below:

1. Object aliases rename objects
    - constant
    - signal
    - variable
    - file
2. Non-object aliases rename items that are not objects
    - function names
    - literals
    - type names
    - attribute names

[Bhasker95]

IEEE VHDL LRM

Aliases

An Example

VHDL HOME PAGE

- An alias of an overloaded subprogram or literal requires a signature to determine the correct value to return

```
TYPE mvl IS ('U', '0', '1', 'Z');
TYPE trinary IS ('0', '1', 'Z');

ALIAS mvl0 IS '0' [RETURN mvl];
ALIAS tri0 IS '0' [RETURN trinary];
```

```
PROCEDURE preset_clear (SIGNAL drv: mvl_vector;
    pc_value: INTEGER);
PROCEDURE preset_clear (SIGNAL drv: bit_vector;
    pc_value: INTEGER);

ALIAS pcmvl IS preset_clear(mvl_vector, INTEGER);
ALIAS pcbit IS preset_clear(bit_vector, INTEGER);
```

Foreign Interfaces

- VHDL provides for some use of foreign languages (e.g. C, Fortran, ADA, etc.)
  - Subprogram or architecture body can have non-VHDL implementation
  - Designer has access to previously written code or code difficult to write in VHDL
- The use of foreign code is mainly implementation dependent
- Foreign variables, signals, or entities are not possible

```
ATTRIBUTE FOREIGN OF name: construct IS
">information/parameters";
```

[Berge93]

IEEE VHDL LRM

Records

Data Types

VHDL HOME PAGE

- The ADEPT token consists of several data types

```
TYPE handshake IS (removed, acked, released, present);
```

- Records can contain records within them

```
TYPE color_type IS
    RECORD
      tag1    : integer;
      tag2    : integer;
      .
      .
      tag15   : integer;
      boole1  : boolean;
      boole2  : boolean;
      boole3  : boolean;
    END RECORD;
```


Advantages of Object Oriented VHDL

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

As an example of a system level modeling methodology using a VHDL-based framework, the object-oriented VHDL tools provided by Vista Technologies, Inc. will be presented in this section.

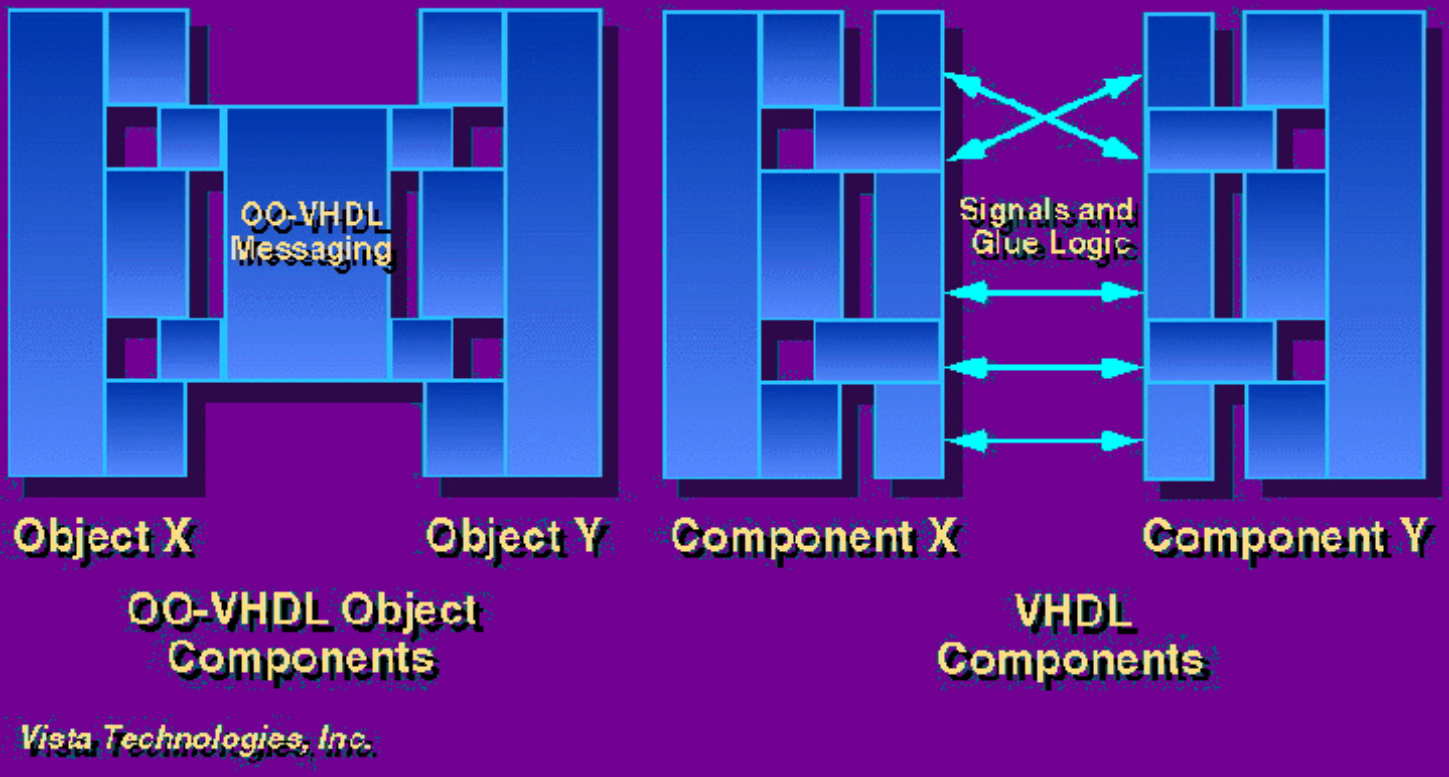
Object-oriented mechanisms allow the interconnection of VHDL components in the model to be generalized and abstracted away. This facilitates experimenting with alternative system architectures.

Advantages of OO-VHDL

(Cont.)

- Benefit: No compatibility Problems between Models
- Enables Interoperability among object components

Any object can send/receive from any other object.
Object communication protocol implicity deifined.



Aliases:

An Example

-- Notes Page --

A signature is required for an alias of a subprogram or an enumeration literal where the *type* of the value returned by alias may not discernible. A signature resolves any ambiguities caused by overloaded subprogram names and overloaded enumeration literals. A signature is indicated by a set of outer brackets, "[" and "]", and shows the appropriate *type*.

[Bhasker95]

Foreign Interfaces

IEEE VHDL LRM

-- Notes Page --

VHDL HOME PAGE

VHDL allows the functionality of architecture bodies and subprograms to be described in a foreign language (e.g. C) and interfaced to a VHDL model. For example, foreign code may be used when it is difficult to implement the same functionality in VHDL, such as in cases where complex arithmetic functions not directly available in VHDL are required.

The interface between VHDL and foreign code is simulator implementation dependent. VHDL passes the parameters to the foreign code but has no further information about the foreign code. The use and structure of foreign code is largely up to the simulator implementation.

Foreign Interfaces

An Example

IEEE VHDL LRM

VHDL HOME PAGE

```
ENTITY and_2input IS
     PORT (a, b: IN BIT;
           c: OUT BIT);
END and_2input;

ARCHITECTURE foreign_model OF and_2input IS
     ATTRIBUTE FOREIGN OF foreign_model:
       ARCHITECTURE IS ffand_2input(A, B, C);
BEGIN
END foreign_model;
```

- The foreign_model architecture is declared as FOREIGN
  - No statements are needed in the architecture body as they will never be executed
  - The implementation calls the ffand_2input function to perform the actions for the and_2input entity

IEEE VHDL LRM

Text Input and Output

VHDL HOME PAGE

- Basic file operations in VHDL are limited to unformatted input/output
- VHDL provides the TEXTIO package for input and output of ASCII text
  - TEXTIO is included in STD library
- The following data types can be used by TEXTIO:
  - Bit, Bit_vector
  - Boolean
  - Character, String
  - Integer, Real
  - Time

IEEE VHDL LRM

Records

VHDL HOME PAGE

- Records are used to collect one or more elements of different types in a single construct
- Records are often used in uninterpreted modeling to represent tokens
- The token record type shown here is from ADEPT

```
TYPE token IS
    RECORD
      status : handshake;
      color : color_type;
      mon_color : mon_color_type;
    END RECORD;
```

Records:

Data Types

-- Notes Page --

The *handshake* data type is an enumerated type consisting of four values. ADEPT uses a fully interlocked, four cycle handshaking scheme for data exchange. Once a token is *placed* by a source, it must be *acked* by a sink. It will then be *released* by the source, and subsequently it will be *removed* by the sink to complete the cycle.

Additionally, tokens can have *color*. The color is defined in a variety fields in a record of type *color_type*. The color fields can represent data length, data type, destination address, source node addresses, or any of a number of other forms of useful information.

[UM93]

Advantages of OO-VHDL (Cont.)

-- Notes Page --

The OO-VHDL paradigm leads to component descriptions that are modular and easily interconnected. In addition, the tool automatically generates the appropriate structural VHDL descriptions so that the user need not be concerned by amending component entities and testbenches.

Advantages of OO-VHDL

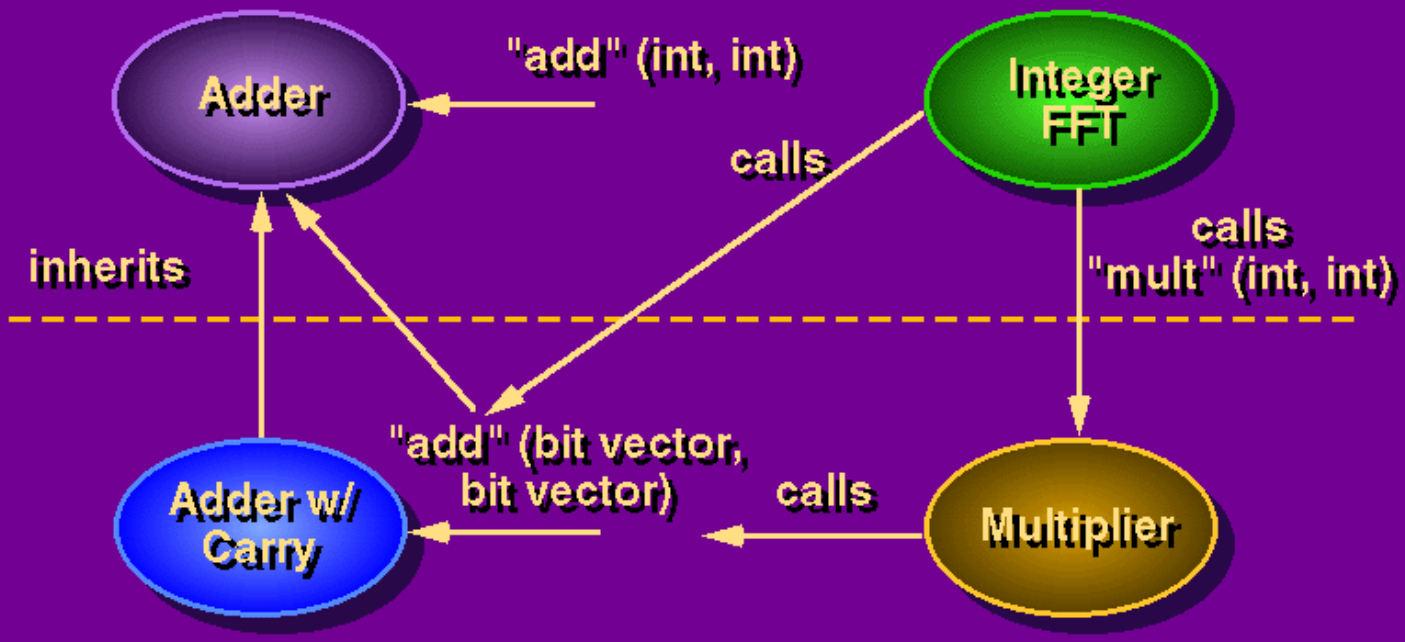
(Cont.)

IEEE VHDL LRM

VHDL HOME PAGE

- Benefit: Can Interface Models with Different Levels of Abstraction
- Enables: Mixed-level modeling

Operations may be redefined as each model increases in detail. Polymorphism allows an algorithm (using operations) written for objects of class X to work on all subclasses of X.



Vista Technologies, Inc.

Foreign Interfaces:

An Example

-- Notes Page --

The foreign model for ffand_2input in this example will be used to provide the functionality of the architecture body. This code could be in a library of other models written in the foreign programming language that may be similarly accessed. An example of a foreign programming language is the 'C' language.

[Bhasker95]

Text Input and Output

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The TEXTIO package provides declarations and subprograms for file and text handling in VHDL. For example, the basic READ and WRITE operations of the FILE type are not very useful. Therefore, the TEXTIO package provides subprograms for manipulating text more easily and efficiently.

IEEE VHDL LRM

TEXTIO Procedures

VHDL HOME PAGE

- TEXTIO defines a LINE data type
  - All read and write operations use the LINE type
- TEXTIO also defines a FILE type of TEXT for use with ASCII text
- Procedures defined by TEXTIO are:
  - READLINE(f, k)
    - reads a line of file f and places it in buffer k
  - READ(k, v,...)
    - reads a value of v of its type from k
  - WRITE(k, v,...)
    - writes value v to LINE k
  - WRITELINE(f, k)
    - writes k to file f
  - ENDFILE(f) returns TRUE at the end of FILE

IEEE VHDL LRM

Assert Statements

VHDL HOME PAGE

- The ASSERT statement is used for displaying text when certain conditions are met
- ASSERT statement classifies the text message in four categories
  - Note -- relays information about conditions to the user
  - Warning -- alerts the user to conditions that are not expected, but not fatal
  - Error -- relays conditions that will cause the model to work incorrectly
  - Failure -- alerts the user to conditions that are catastrophic

Records

-- Notes Page --

VHDL *record* data types contain elements of different VHDL *types*. Records are useful in system level modeling because they allow encapsulation and abstraction to be done easily in the system description. The example shown in this slide is found in [UM93] and presents a record named *token* in which one field (*status*) is used in an associated VHDL *bus resolution function* and the other field (*color*) carries data.

Advantages of OO-VHDL (Cont.)

-- Notes Page --

Components at multiple levels of abstraction can be interconnected in a single model. This slide also shows another aspect of the OO-VHDL paradigm where polymorphism is supported to facilitate reuse of *subprograms* and *types*.

Advantages of OO-VHDL

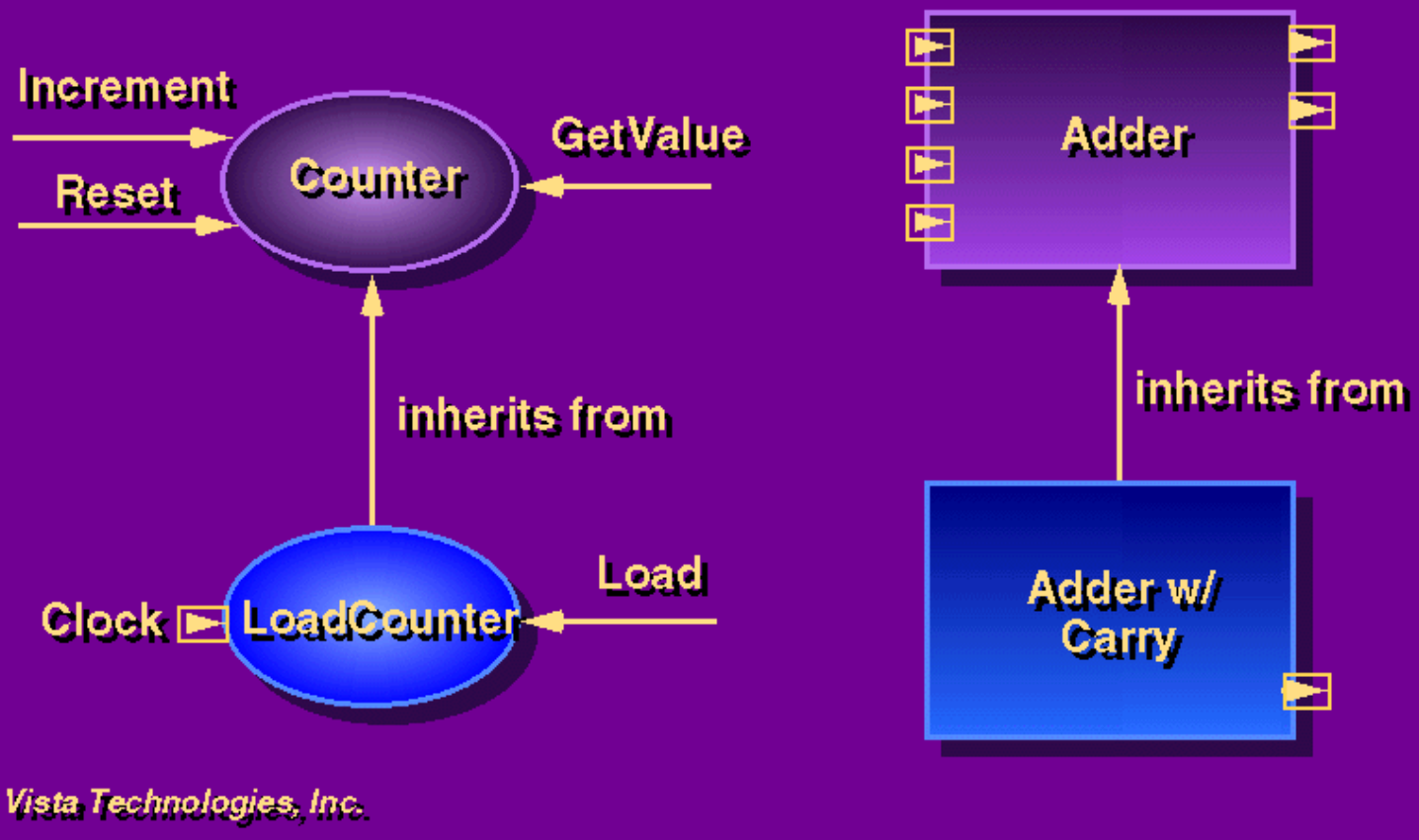
(Cont.)

IEEE VHDL LRM

VHDL HOME PAGE

- Benefit: Can Easily Modify Parts of Behavior
- Enables: Reuse by defining components in terms of other components

Reduces maintenance costs.
Reduces code bulk.
Clarifies similarities and differences among related components.




Map
Notes
Help

IEEE VHDL LRM

Synthesizing OO-VHDL

VHDL HOME PAGE

- High-level features of OO-VHDL prevent it from being synthesized directly

Converting OO-VHDL to Synthesizable VHDL

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This slide lists the steps necessary in removing the abstractions inherent in OO-VHDL so that synthesizable VHDL can be generated.

UVA ADEPT Primitive Modules

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

ADEPT is a performance and reliability analysis tool which provides several types of primitive modules. Every module has both a VHDL and a Petri Net description. While ADEPT can hide the VHDL from the designer, the VHDL descriptions are used in the simulation-based analyses performed using ADEPT.

[UM93]

ADEPT Token Handling

- Defined in "token_definition.pkg"

```
-- (C) 1993 UVA Center for Semicustom Integrated Systems

-- Used for dependent outputs and data inputs
PROCEDURE place_token (SIGNAL T: INOUT token; tag: color_type;
                       mon_tag: mon_color_type; delay: time);
PROCEDURE ack_token (SIGNAL T: INOUT token);
PROCEDURE release_token (SIGNAL T: INOUT token);
PROCEDURE remove_token (SIGNAL T: INOUT token);

-- Used for writing to independent outputs and control inputs
PROCEDURE place_control_token (SIGNAL T: OUT token;
       tag: color_type; mon_tag: mon_color_type; delay: time);
PROCEDURE release_control_token (SIGNAL T: OUT token);
PROCEDURE release_control_token_with_color
      (SIGNAL T: OUT token;
        tag: color_type; mon_tag: mon_color_type);
```


Honeywell PML

- The Honeywell Performance Modeling Library (PML) is another approach to performance modeling
  - PML is a token queuing network
  - It does not have a rigid underlying mathematical model
  - Tokens carry the minimal functional information necessary to model systems
- Features of PML
  - Library of over 50 generic components
  - VHDL model automatically keeps track of utilization, latency, and throughput
  - Defines a standard token type

[Honeywell95]

TEXTIO Procedures

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

TEXTIO defines two new data types to assist in text handling. The first is the LINE data type. The LINE type is a text buffer used to interface VHDL I/O and the referenced file. Only the LINE type may read from or written to a file.

The second is the FILE type of TEXT. A file of type TEXT may only contain ASCII characters.

Several of the procedures provided by TEXTIO for handling text input/output are also listed in this slide.

IEEE VHDL LRM

Using TEXTIO

VHDL HOME PAGE

- Reading from a file
  - READLINE reads a line from the file into a LINE buffer
  - READ gets data from the buffer
- Writing to a file
  - WRITE puts data into a LINE buffer
  - WRITELINE writes the data in the LINE buffer to file
- READ and WRITE have several formatting parameters
  - Right or left justification
  - Field width
  - Unit displayed (for time)

IEEE VHDL LRM

TEXTIO

An Example

VHDL HOME PAGE

- This procedure displays the current state of a FSM

```
USE STD.TEXTIO.ALL;
TYPE state IS (reset, good);
PROCEDURE display_state (current_state : IN state) IS
    VARIABLE k : LINE;
    FILE flush : TEXT IS OUT "/dev/tty";
    VARIABLE state_string : STRING (1 to 7);
BEGIN
    CASE current_state IS
        WHEN reset => state_string := "reset ";
        WHEN good => state_string := "good ";
    END CASE;
    WRITE (k, state_string, LEFT, 7);
    WRITELINE (flush, k);
END display_state;
```

Assert Statement

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The ASSERT statement is used to alert the user of some condition inside the model. When the expression in the ASSERT statement evaluates to FALSE, the associated text message is displayed on the simulator console. Additionally, an evaluation of FALSE may halt the simulation depending on the severity level of the associated ASSERT statement.

The four severity levels, in increasing severity, are listed in this slide. However, the definitions of the severity levels are somewhat general and can be modified to suit the designer's needs.

IEEE VHDL LRM

Use of Assert Statements

VHDL HOME PAGE

- Syntax of the ASSERT statement

```
ASSERT condition
    REPORT "violation statement"
    SEVERITY level;
```

- The ASSERT statement will trigger when the condition is *false*
- The violation statement must be enclosed in quotes

```
ASSERT (NOT ((j='1') AND (k='1')))
    REPORT "Set and Reset are both 1"
    SEVERITY ERROR;
```

Abstract Data Types

- To enhance modeling efficiency and usefulness, VHDL supports the notion of abstract data types (ADT's)
- Examples
  - Queue data type
  - Finite state machine data type
  - Floating and complex data types
  - Vector and matrix data types
- An abstract data type consists of two things
  - Data
  - Operators that manipulate the data
- ADT's are implemented in VHDL through the use of a set of components which operate in a consistent integrated data environment

Advantages of OO-VHDL (Cont.)

-- Notes Page --

A new component can also be described as a subclass of a previously described component so that it can inherit much of its behavior from the original component. The subclass component description then only needs to include behavior that is unique to it.

Advantages of OO-VHDL

(Cont.)

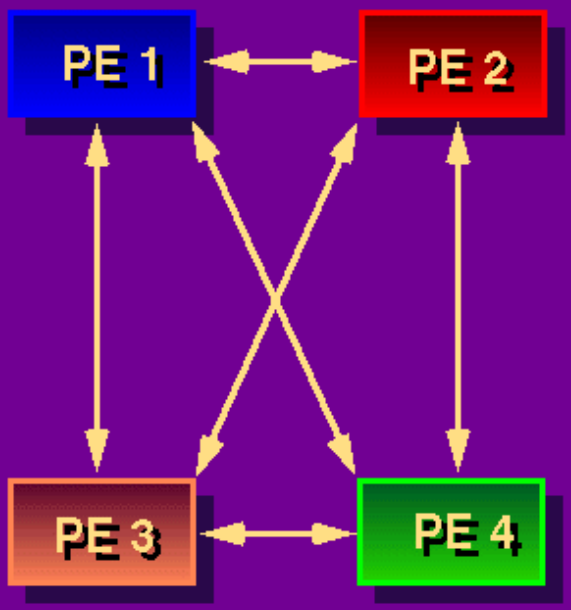
IEEE VHDL LRM

VHDL HOME PAGE

- Benefit: No Port Declarations Necessary
- Enables: Scalable component to component communication

Only "handles" required to communicate with other objects.
Communication pathways managed by OO-VHDL.
Trivial to add one more component to testbench.

OO-VHDL Object Model



VHDL Component Model

Vista Technologies, Inc.

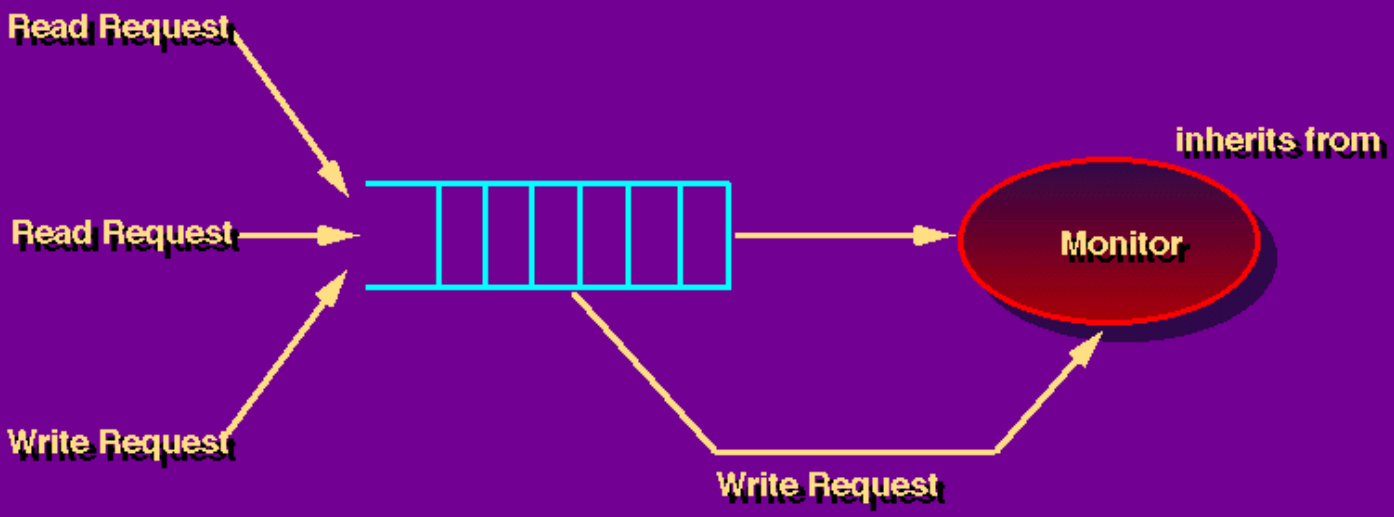

Map
Notes
Help

Advantages of OO-VHDL

(Cont.)

- Benefit: Provides Monitor Capability Today
- Enables: Access control shared resources

Mutual-exclusion provided by OO-VHDL message serialization.
Requesters are blocked until their request is serviced.
Priorities may be placed on monitor operations.



Vista Technologies, Inc.

Synthesizing OO-VHDL

-- Notes Page --

The high-level abstractions used in OO-VHDL models do not allow them to be synthesized directly. For example, a synthesis tool cannot (and should not) conjure an implementation for the abstract communication mechanism used by OO-VHDL components. However, by means of transformation tools directed or specified by the user, the descriptions of OO-VHDL models can be translated into synthesizable VHDL code so that an implementation is generated through a user-directed synthesis process.

UVA ADEPT:

Token Handling

-- Notes Page --

IEEE VHDL LRM

VHDL HOME PAGE

ADEPT has many procedures and functions defined to handle the user-defined data type *token*. Some of the procedures and functions are presented here.

The first four procedures are used for *dependent output* and *data input*. These procedures perform the handshaking necessary for *data tokens* which require a fully interlocked communication protocol between a master and a slave.

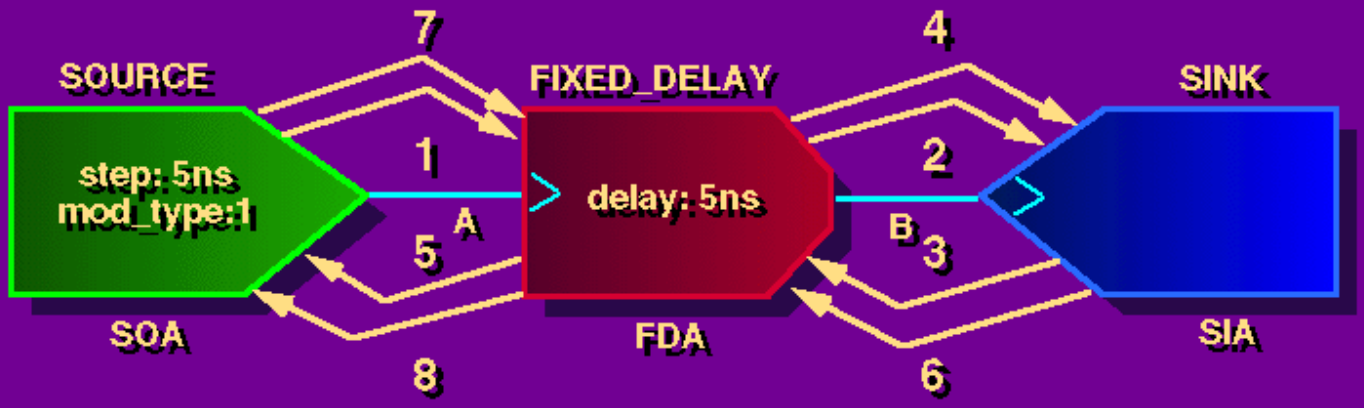
The last three procedures handle *independent* or *control tokens*. *Control tokens* do no handshaking protocol because the master may place and remove them independently of what the slave does with the *tokens*.

[UM93]

ADEPT Token Handling (cont.)


```
-- Used to determine status of input and output token
FUNCTION token_present (T: Token) RETURN boolean;
FUNCTION token_acked (T: Token) RETURN boolean;
FUNCTION token_released (T: Token) RETURN boolean;
FUNCTION token_removed (T: Token) RETURN boolean;
```


Three Module Example Detailed Event Sequence



UVa

Event	Time	Delta	Description	Resolved Signal A	Resolved Signal B
1	0 ns	1	Source executes place_token on A	present	removed
2	5 ns	1	Delay executes place_token on B	----	present
3	5 ns	2	Sink executes ack_token on B	----	acked
4	5 ns	3	Delay executes release_token on B	----	released
5			Delay executes ack_token on A	acked	----
6	5 ns	4	Sink executes remove_token on B	----	removed
7			Source executes release_token on A	released	----
8	5 ns	5	Delay executes remove_token on A	removed	----
1	10 ns	1	Source executes place_token on A	present	----

Honeywell PML

-- Notes Page --

The Honeywell Performance Modeling Library (PML) is another approach that uses VHDL for system level performance modeling. The PML implements a token-based approach. Use of the PML requires a simulation-based analysis because, unlike ADEPT, the PML does not have a rigid underlying mathematical (i.e. Petri Net) model. The PML library, however, offers components with more complex behavior than the ADEPT modules.

[Honeywell95]

Honeywell PML

Token Type

- The PML uinterface_token has several fields related to bus resolution

```
TYPE uinterface_token IS
RECORD

    --  user fields
    parm1_real  :  REAL;                          -- these are
placed first to avoid
    parm2_real  :  REAL;                          -- some oddities
on Sparcs
    parm1_int   :  INTEGER;
    parm2_int   :  INTEGER;

    --  control flow
    destination :  name_type;
    source      :  name_type;
    t_type      :  token_type;

    --  performance fields
    size        :  data_size;
    value       :  INTEGER;

    --  token tracking or statistics fields
    id          :  uGIDType;
    start_time  :  TIME;

    -- communication fields
    priority    :  INTEGER;
    state       :  State_Type;
    protocol    :  Protocol_Type;

    -- user communication tracking and control fields
    collisions  :  INTEGER;
    retries     :  INTEGER;
    route       :  INTEGER;
```

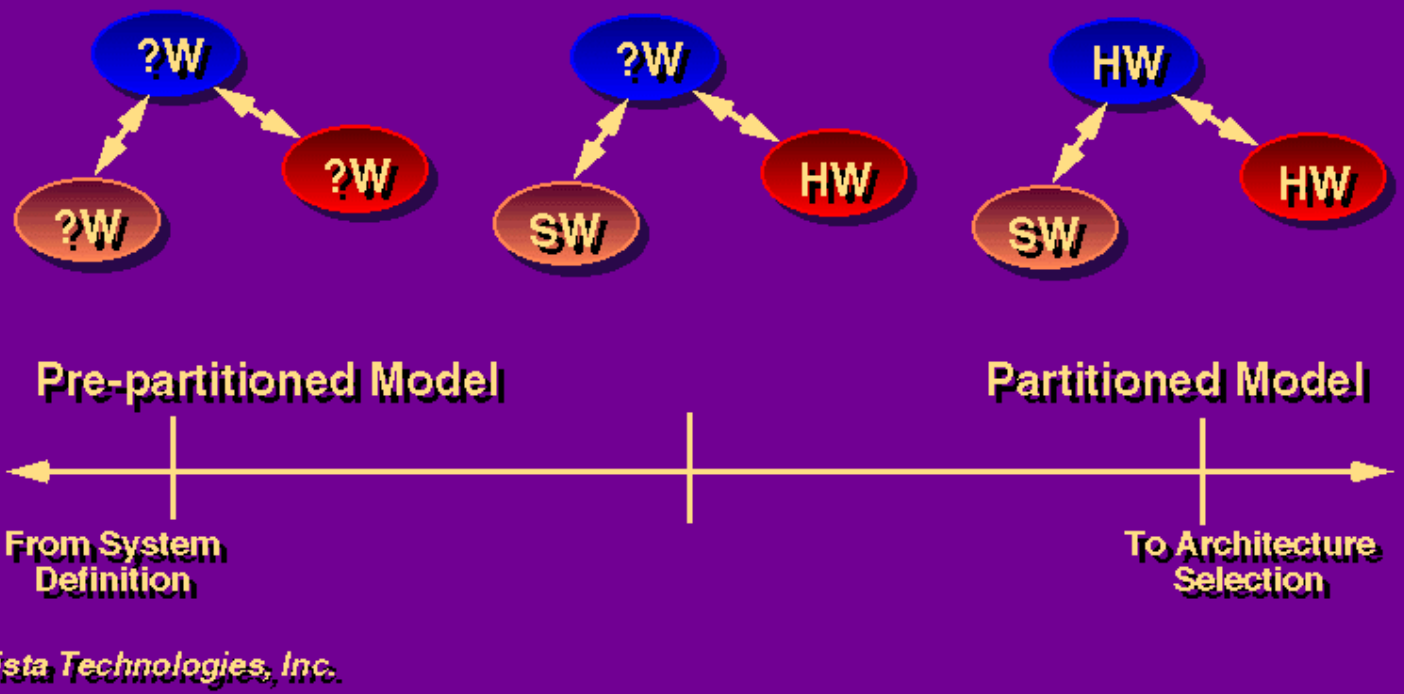
```
END RECORD;
```


Advantages of OO-VHDL

(Cont.)

- Benefit: HW and SW Object Communicate as Equals
- Enables: Hardware/Software Codesign

Behavior is described in terms of operations, not just port interfaces.
Single model may contain HW objects, SW objects, or "pre-partitioned" objects.
OO-VHDL messages provide a common communications medium.



Advantages of OO-VHDL

(Cont.)

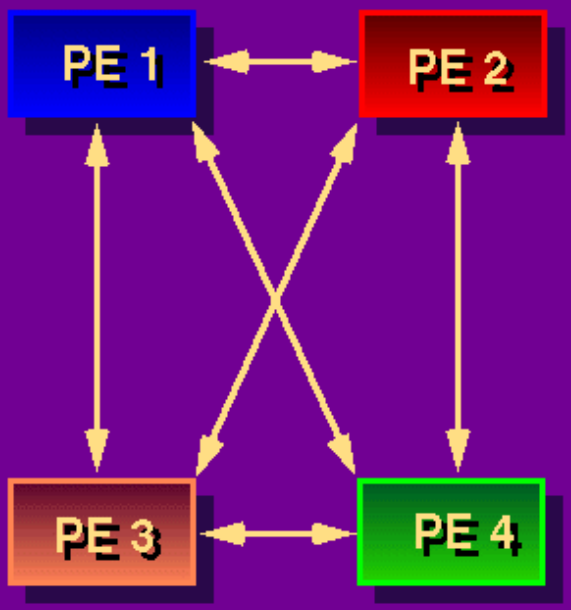
IEEE VHDL LRM

VHDL HOME PAGE

- Benefit: No Port Declarations Necessary
- Enables: Scalable component to component communication

Only "handles" required to communicate with other objects.
Communication pathways managed by OO-VHDL.
Trivial to add one more component to testbench.

OO-VHDL Object Model



VHDL Component Model

Vista Technologies, Inc.


Map
Notes
Help

Advantages of OO-VHDL (Cont.)

-- Notes Page --

The standard interface mechanism provided by OO-VHDL allows functionality which is implemented as software to be described as a VHDL component and interconnected with VHDL components describing hardware.

Advantages of OO-VHDL (Cont.)

-- Notes Page --

OO-VHDL provides monitor components which can be placed in the model to facilitate the tracking and recording of model status information during simulations.

ADEPT Token Handling (cont.)

-- Notes Page --

These four functions are used to check on the value of the *status* field of the *token* being examined. The use of these functions provides a level of abstraction with regard to how the *token* structure is implemented in ADEPT.

[UM93]

Basic Module Format Packages and Entity

IEEE VHDL LRM

VHDL HOME PAGE

```
-- (C) 1993 UVA Center for Semicustom Integrated Systems -- [1]
LIBRARY package_defs; USE package_defs.basic_defs.ALL; USE
package_defs.token_definition.ALL; USE
package_defs.color_fcns.ALL; USE
package_defs.mon_color_fcns.ALL; ENTITY module_name IS -- [2]
GENERIC (generic_list: generic_types); -- [3] PORT (data_input :
INOUT token; dependent_output : INOUT token; control_input : IN
token; independent_output : OUT token); END module_name;
```


Basic Module Format:

Packages and Entity

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The following slides present the standard format for an ADEPT module. Adherence to this standard format facilitates the maintenance of the ADEPT library of modules.

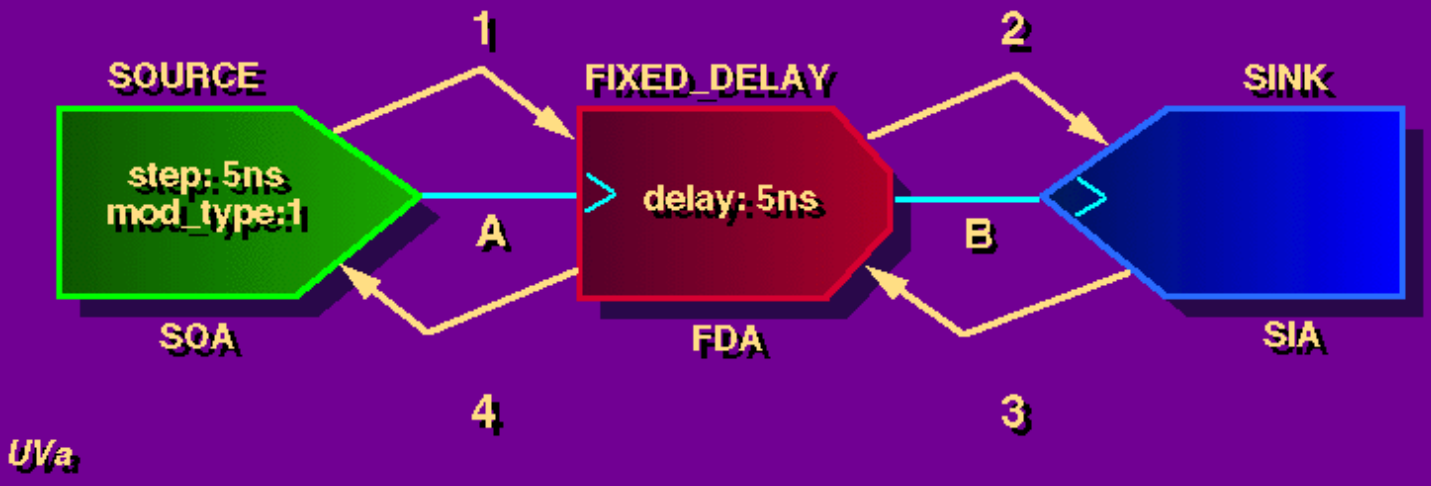
ADEPT *type* and *subprogram* definitions are provided in a library called *package_defs*. Several packages within that library are available. In ADEPT, the generics are often specified on the schematic to configure instantiated modules. The *data_input* and *dependent_output* ports must be of INOUT type because these *tokens* use the four-cycle communication protocol. The control_input and independent_output are IN and OUT, respectively, since they are only driven by the *token* masters.

[UM93]

Basic Module Format
Architecture


```
ARCHITECTURE ar_module_name OF module_name IS
BEGIN
process_1:PROCESS
   -- [4]
   VARIABLE variable_names : variable_types;
   BEGIN
   -- [5]
   IF token_present (data_input) AND
     token_removed (dependent_output) THEN
     -- do something
     place_token (dependent_output, color_of (data_input),
                  mon_color_of (data_input), delay);
   END IF;
   -- [6]
   IF token_present (control_input) THEN
   -- do something
   place_control_token (independent_output,
    color_of (control_input), mon_color_of (control_input, 0 ns));
```


Three Module Example Simplified Event Sequence



Event	Time	Description
1	0 ns	Source places token on A
2	5 ns	Delay places token on B
3	5 ns	Sink acks token on B
4	5 ns	Delay acks token on A
1	10 ns	Source places next token on A

Three Module Example:

Detailed Event Sequence

-- Notes Page --

This example shows the entire four-cycle sequence of token assignments made in the passing of *tokens* in the model. Note that after a *token* is acknowledged, the release and removal of that *token* take place in delta time (e.g. event 4 and 6 for *B* in the example).

[UM93]

Honeywell PML:

Token Type

-- Notes Page --

The Honeywell PML defines a standard token named *uinterface_token*. The token has several fields, including some used by its associated bus resolution function. The *protocol* field can be used to implement different protocols in the system. The *priority*, *collisions*, *retries* and *interrupt* fields are used by the bus resolution function.

This token definition is from the RASSP VHDL Token-based Performance Modeling Interoperability Guideline, Version 2.0. The current public release version of this document is available at http://rassp.scra.org/public/tb/honeywell/HONEYWELL-DOCS.html.

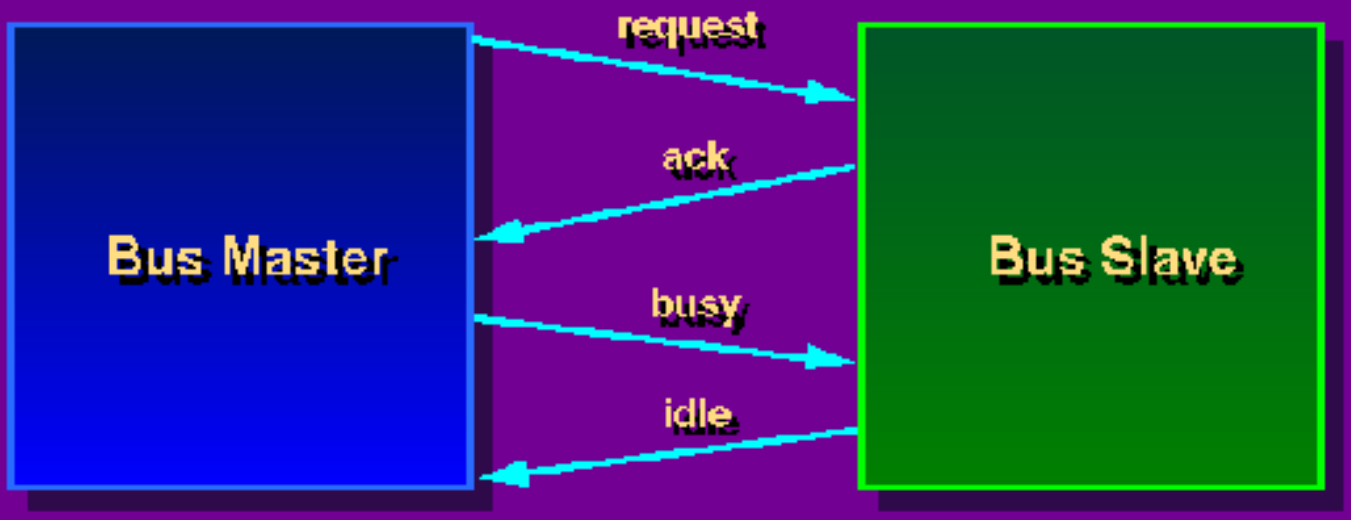
[Honeywell95]

Honeywell PML Handshaking Protocol

IEEE VHDL LRM

VHDL HOME PAGE

- Honeywell PML uses a handshaking protocol similar to that of ADEPT
- Tokens on the bus have one of four values: idle, request, ack, busy
- Tokens change the state of the token by changing the state field of the utoken RECORD



UVA ADEPT Bus Resolution

IEEE VHDL LRM

VHDL HOME PAGE

- This table shows the bus resolution between the source, sink, and fixed delay in this three module example

Dependent Output	Data Input	Resolution
token released	token removed	token removed
token present	token removed	token present
token present	token acked	token acked
token released	token acked	token released

Three Module Example:

Simplified Event Sequence

-- Notes Page --

The three-module example shown in this slide will be used to illustrate the important events in the passing of tokens from a token master to a token slave. As the table shows, the relevant events are the *placing* and *acknowledging* of tokens. The other two states in the four-cycle handshake, *releasing* and *removing*, are only required to effect the interlock in a shared medium.

Note that the *fixed_delay* module does not acknowledge the *source's token* until its output has been acknowledged by the *sink* module (i.e. there is no buffering between inputs and outputs). This is an important characteristic of the communication standard used by ADEPT modules (unless explicitly stated otherwise, as in the BUFFER module).

[UM93]

Honeywell PML:

Handshaking Protocol

-- Notes Page --

The sequence of token states used by the PML utoken is illustrated here. The sequence is similar to the one used by ADEPT, but because the PML supports multiple masters simultaneously (with different priorities to differentiate them), the bus resolution function is more complex than in ADEPT.

[Honeywell95]

Honeywell PML Bus Resolution

- Bus resolution function (BRF) resolves a signal value that has multiple drivers
- BRF and bus interface unit (BIU) characterize the bus
- Resolution function arbitrates based on token fields
  - bus state
  - protocol
  - priority
  - token id
- BRF may need to be characterized for different protocols

Honeywell PML

Steps to Set-up the Functional Memory


```
LIBRARY GEN;
USE GEN.dataio.IPC_real_type;

LIBRARY PROC;
USE PROC.ioport_signal.IOPort;

ARCHITECTURE testbench OF fivecpu_testbench IS
  COMPONENT Processor
    PORT (DataPort : INOUT IPC_real_type);
  END COMPONENT;

  COMPONENT Func_Memory_Real
      PORT (DataPort : INOUT IPC_real_type);
  END COMPONENT;
  FOR CPU_1: Processor USE
    CONFIGURATION PROC.PROCESSOR_01;

  FOR MEM: Func_Memory_Real USE ENTITY
    proc.Func_Memory_Real (system);

  SIGNAL token_interconnect : token;
```

```
BEGIN
 MEM:Func_Memory_Real
   PORT MAP (DataPort => IOPORT);
 CPU_1:Processor
   PORT MAP (Data0=> token_interconnect);
```

not a complete code example

- First, make an instance of the memory
- In the modules that require access to the Functional Memory, include the ioport_signal USE clause to instantiate the global signal
  - USE proc.ioport_signal.ioport, or
  - USE proc.ioport_signal.ioport_i
  - No explicit port declarations for the other modeling artifacts are necessary

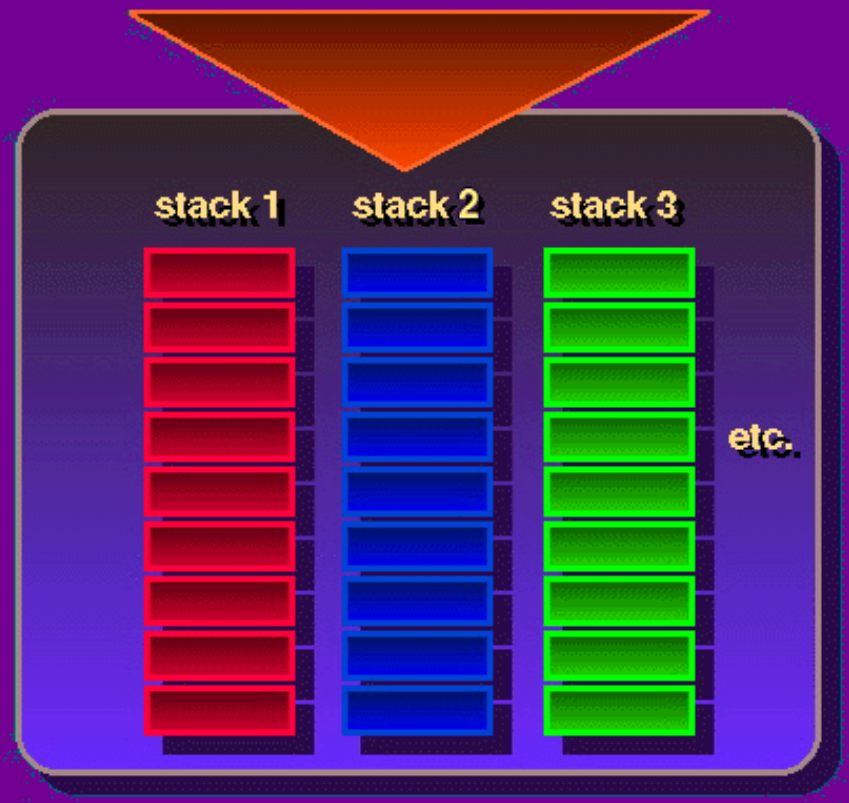
Honeywell PML Functional Memory Interface

IEEE VHDL LRM

VHDL HOME PAGE

```
ENTITY Func_Memory IS
  PORT (DataPort : INOUT IPC_real_type);
END Func_Memory;

ENTITY Func_Memory IS
  PORT(DataPort : INOUT IPC_integer_type);
END Func_Memory;
```



Honeywell

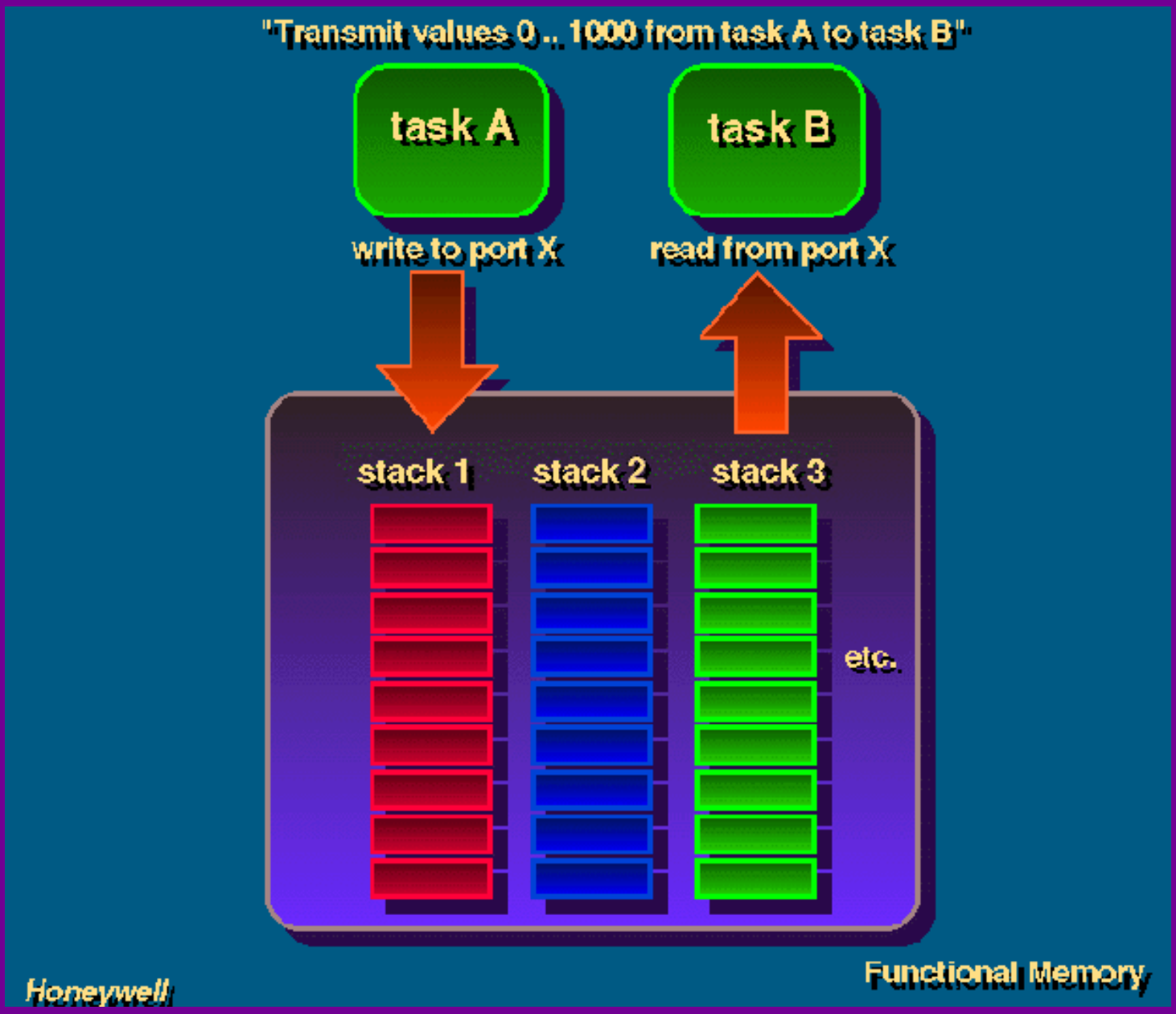
Functional Memory

- Two data types supported: Integers and Reals
- Data written/read from Functional memory are vectors (integers or reals)
  - Signal types defined in the base_types package of the GEN library
  - The bus resolution function will handle all arbitration
  - Complex data types are supported by user-defined wrapper procedures (need to convert complex data types into groups of integer or reals)

IEEE VHDL LRM

Honeywell PML Functional Memory Component

VHDL HOME PAGE



- Situations arise when the user will want to model the functional flow of data between processor model components
  - More detailed functionality desired

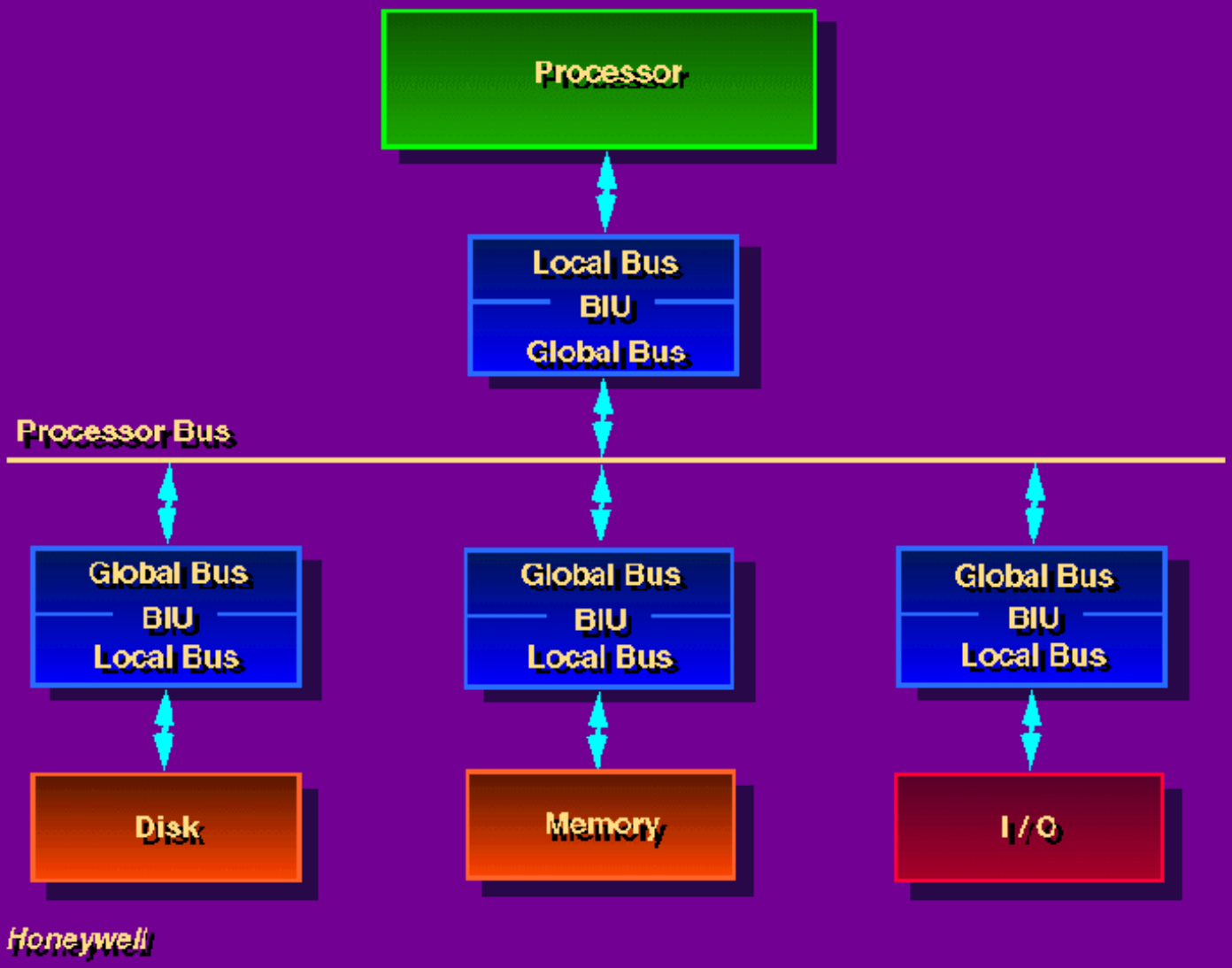
- Performance modeling requires application dependent behavioral information
- Modeling control flow
- Test boundary conditions

- Functional Memory component allows software tasks to exchange large amounts of data without impacting simulation time (no resource utilization)
  - Supports arbitrary sizes of data
  - Simple interface
  - Not a huge burden on simulation runtime performance
  - Persistent; no synchronized transmitters/receivers needed

IEEE VHDL LRM

Honeywell PML
Use of BIU

VHDL HOME PAGE



Honeywell PML:

Functional Memory Component

-- Notes Page --

A configurable Functional Memory Component is provided for storage of desired model information. It provides a simple interface and supports the ability to communicate large amounts of information from one component to another efficiently by providing a shared memory space.

[Honeywell95]

Honeywell PML:

Functional Memory Component

-- Notes Page --

Two types of Functional Memory Components are provide; one for the storage of integer data, and another for the storage of real (i.e. floating point) data. Other forms of data require the use of builtin or user-defined functions for conversion to type *integer* or type *real* for storage.

[Honeywell95]

Honeywell PML:

Steps to Set-up the Functional Memory Operations

-- Notes Page --

This slide shows an example instantiation of a Functional Memory Component.

[Honeywell95]

Honeywell PML:

Functional Memory Operations

-- Notes Page --

This slide shows the parameters used in the read and write procedures provided to interface with Functional Memory Components easily. Note that the procedures support selection from among several memory components as well as accesses of vector data in one operation.

[Honeywell95]

IEEE VHDL LRM

Using TEXTIO

-- Notes Page --

VHDL HOME PAGE

TEXTIO requires that all disk access go through a buffer of type LINE. In addition, the READ and WRITE procedures can further format the text. The field width of the text is its length if not otherwise specified. If the text is of type TIME, the unit of time can be specified.

TEXTIO:

An Example

-- Notes Page --

IEEE VHDL LRM

VHDL HOME PAGE

This example displays the current state of a finite state machine model execution.

First, the USE clause makes the contents of the TEXTIO package available. The enumerated type STATE is also declared. The procedure *display_state* requires only one input value, the current state of the FSM.

Several local variables are declared. The buffer *k* of type LINE will be used for WRITE storage. The FILE *flush* is of type text and will output to a file named */dev/tty*. This logical name is the standard output or screen in UNIX. Therefore, the procedure will write to the screen. The variable *state_string* holds the text value of the state.

The CASE statement converts the state of the FSM into a text value. The WRITE statement then writes the value of *state_string* to the buffer *k*. The WRITE statement further specifies that the string should be left justified and be 7 spaces wide.

Finally, the WRITELINE sends the buffer *k* to the file *flush*. The text is then written to the screen.

Note that this procedure would not work very well for writing to a file. Since the file is re-initialized every time the procedure is used, the text would always be written to the beginning of the file. Using TEXTIO to write to a file requires that the file be passed to the procedure as a

parameter, or the modeler could use a process that implements the same functionality.

Based on [Navabi93]

Assert Statements

-- Notes Page --

This slide shows the syntax of the ASSERT statement. The ASSERT statement will trigger when the condition is false. The REPORT statement to be displayed is enclosed in quotes.

The *Set* and *Reset* lines of the J-K flip-flop in this example cannot simultaneously equal one. Therefore, the ASSERT statement evaluates to FALSE if this situation is observed during simulation.

IEEE VHDL LRM

Assert Statements

An Example

VHDL HOME PAGE

- This code has similar functionality to that of the TEXTIO example
  - Assume good = '1', reset = '0'

```
PROCEDURE display_state (current_state : IN state) IS
BEGIN
   ASSERT NOT(current_state = '1')
      REPORT "Status of State: good"
      SEVERITY NOTE;
   ASSERT NOT(current_state = '0')
      REPORT "Status of State: reset"
      SEVERITY WARNING;
END display_state;
```

- ASSERT statements may have some implementation defined action associated with the various SEVERITY levels

Abstract Data Types in VHDL

-- Notes Page --

The discussion on Abstract Data Types is adapted from: Sidhatha Mohanty, V. Krishnaswamy, P. Wilsey, "Systems Modeling, Performance Analysis, and Evolutionary Prototyping with Hardware Description Languages," Proceedings of the 1995 Multiconference on Simulation, pp 312-318.

Abstract data types (ADTs) are objects which can be used to represent an activity or component in behavioral modeling. An ADT supports data hiding, encapsulation, and parameterized reuse. As such they give VHDL some object-oriented capability.

An ADT is both a data structure (such as a stack, queue, tree, etc.) and a set of functions (i.e. operators) that provide useful services on the data. For example, a stack ADT would have functions for pushing an element onto the stack, retrieving an item from the stack, and perhaps several user-accessible attributes such as whether the stack is full or empty.

Abstract Data Types

An Example

- Data structure: a queue with user-defined attributes
- Operators
  - Token source
  - Fork and join
  - Server
  - Sink
- To use the queue ADT, set several generic parameters to define the sizes, distributions, and other characteristics provided by the ADT
- Example - to define and use a queue

Advantages of OO-VHDL (Cont.)

-- Notes Page --

The benefit of using an object-oriented paradigm in VHDL modeling is that it leads to compact system descriptions which can be modified easily to facilitate maintainability and complexity management.

Advantages of OO-VHDL (Cont.)

-- Notes Page --

The OO-VHDL mechanism abstracts away interconnection details so that various alternative structural descriptions can be described easily.

Basic Module Format:

Architecture

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The basic architecture is shown here and in the next slide.

Note that the basic behavior of a module first requires that a *token* (or *tokens*) be recognized as *present* on some relevant input(s). This may be some prescribed combination of *data* or *control* inputs. Once the appropriate input condition is recognized, the module examines the relevant output port to ensure that it is available (i.e. the four-cycle handshake from any previous communication has completed). If the required output port is available, the module will then perform its function and place a *token* on the output.

[UM93]

Basic Module Format
Architecture (cont.)


```
ELSE
   -- no input token, release token from independent output
   release_control_token (independent_output);
END IF;
-- [7]
IF token_acked (dependent_output) THEN
   -- Pass output acknowledgement back through
   release_token (dependent_output);
   ack_token (data_input);
END IF;
-- [8]
IF token_released (data_input) THEN
   remove_token (data_input);
END IF;
-- [9]
WAIT ON data_input, dependent_output, control_input;
END PROCESS;
END ar_module_name;
```


IEEE VHDL LRM

Three Module Example

VHDL HOME PAGE

- This example shows the event sequence in a simple three module example

UVa

Time between new tokens on A=step+path_delay= 10 ns

UVA ADEPT Bus Resolution

-- Notes Page --

This table illustrates the possible conditions of the status fields for both the master *token* driver and the slave *token* driver. The bus resolution function (BRF) implements the priorities shown in this table. For example, if the master drives a token with status *present* while the slave drives a token with status *acked*, the BRF returns the token with status *acked*.

[UM93]

Honeywell PML:

Bus Resolution

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The BRF for the PML is versatile and its returned *utoken* is based on the values on four fields of the driven *utokens* (*state*, *protocol*, *priority*, and *id*).

[Honeywell95]

Honeywell PML Bus Resolution Code

IEEE VHDL LRM

VHDL HOME PAGE

```
-- (C) 1995 Honeywell Inc.  All rights reserved.
BEGIN
  -- If no drivers, just drive initial token
  IF (s'LENGTH <1) THEN
    RETURN init_token;
  -- If one input, default is single driver
  ELSIF (s'LENGTH = 1) THEN
    RETURN s(s'LOW);
  ELSE
    -- Ensure all drivers have the same protocol
    current_protocol := s(s'LOW).protocol;
    poll_drivers: FOR i IN s'RANGE LOOP
      CASE current_protocol IS
```

Honeywell PML Bus Interface Unit

IEEE VHDL LRM

VHDL HOME PAGE

- The BIU consists of four processes:
  - LBRCV receives tokens from local bus
  - LBTX transmits tokens on the local bus
  - BIURCV receives tokens from the global bus
  - BIUTX transmits tokens on the global bus
- The BIU is a generic module that performs handshaking for other modules
  - Shields modules from details of the bus

Honeywell PML:

Use of BIU

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This slide shows an example use of the BIU. The BIU is placed between the global bus and the local bus of the module. Different modules may have different handshaking or timing requirements; the BIU is used to satisfy these requirements and those of the shared bus.

[Honeywell95]

Assert Statements:

-- Notes Page --

The example shown here provides a similar functionality to the TEXTIO example shown previously. The ASSERT statements are used to display the current state of an FSM. Note that these ASSERT statements are concurrent. ASSERTs can be concurrent or sequential depending on whether they appear inside or outside VHDL *processes*. In fact, because they are passive statements (i.e. no assignments are made) ASSERT statements can also be put in VHDL ENTITY statements.

In the example here, if the "good" state is defined to be 1, then the first ASSERT will trigger when current_state is not equal to 0. The second ASSERT statement is set up in a similar fashion.

While this mechanism is similar to the previous TEXTIO example, it can provide more information to the user and the simulator. The SEVERITY level may cause the simulator to pause or stop altogether. While these definitions are implementation defined, they can be useful.

Abstract Data Types:

An Example

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

Consider a queue abstract data type. It will have a 1) data structure for holding tokens and 2) functions for inserting/retrieving tokens in/from the data structure.

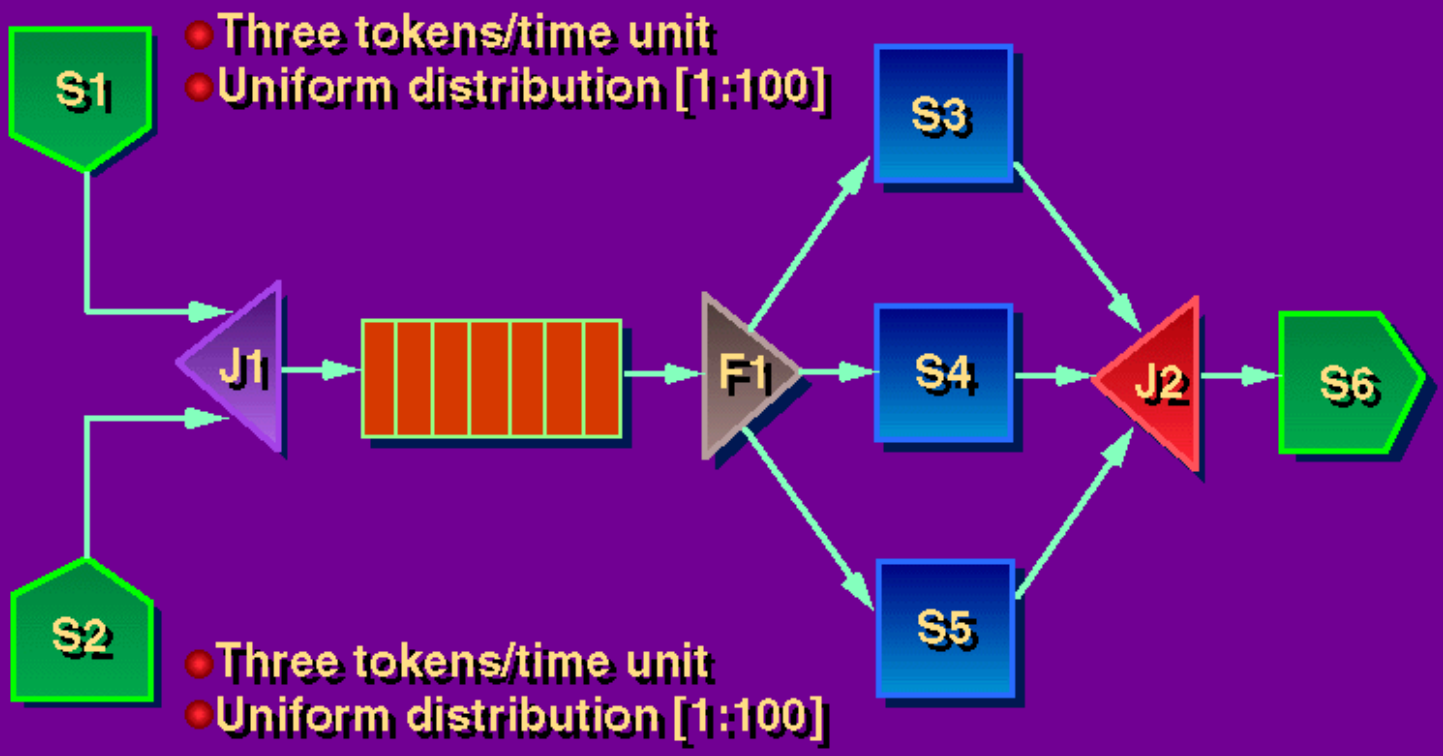
In fact, since a queue is used within a queuing system, we will expand our definition of operators to include 1) a generator for tokens, 2) a salvager of tokens, 3) a server, and 4) fork and join operators.

Thus, our queuing system ADT can be used to create an entire system of queues, servers, and token generators to model a system (e.g. a DSP) at an uninterpreted (i.e. non-functional) level.

IEEE VHDL LRM

Example Use of QUEUE ADT

VHDL HOME PAGE



Map Notes Help

Example Use of QUEUE ADT

-- Notes Page --

As an example, consider a queuing network which is a high-level model for a 3-CPU multiprocessing computer on a single bus and two ports to memory. The queue represents the serializing of memory data entering the bus.

S1 and *S2* are identical token sources each supplying three tokens at a time with inter-arrival times uniformly distributed between 1 and 100 time units. *J1* receives all generated tokens and provides a single stream of tokens to the single queue. Tokens are removed from the queue whenever any of the three servers, *S3*, *S4*, and *S5*, is free. *J2* is a join which then receives tokens from the servers and destroys them, thus freeing up the small amount of memory used by a token.

IEEE VHDL LRM

Queue System VHDL Model

VHDL HOME PAGE

```
LIBRARY queue;
USE queue.queue_pkg.all;
USE std.textio.all;

ENTITY open_system is
END open_system;
```

```
ARCHITECURE example OF open_system IS

  SIGNAL arc1, arc2, arc3, arc4, arc5, arc6,
         arc7, arc8, arc9, arc10, arc11 : arc bus;
                  .
                  .
          <component declarations>
                  .
                  .
  BEGIN
           <architecture>
  END example;
```

Queue System VHDL Model

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

In this and the next few slides, we will describe the queuing system ADT and show how to use the ADT to create and exercise the model presented in the previous slide.

The *queue_pkg* package contains a number of functions useful for the queuing system component including overloaded *read* and *write* procedures.

Since we assume there are no system inputs and outputs, the entity open_system has no ports.

The architecture has three basic parts: 1) a signal list declaration which declares all signals used to connect the components together to create the queuing network, 2) the declarations of the queuing system components (i.e., sources, sink, forks, joins, queue, and servers), and 3) the body which here will be little more than a netlist of components.

Queue Example Declarations

```
COMPONENT source1
  GENERIC (out_control : distrib;  seed : integer;
        generation    : frequency; name : string;
        debug_control : debug := none;
        report_freq   : time := 10fs);
    PORT (out_arc : inout arc bus);
  END COMPONENT;
```

```
COMPONENT queue1
  generic (queue_length : integer;
  name : string;
  discipline    : queue_discipline;
  debug_control : debug := none;
  report_freq   : time := 10 fs);

  PORT (in_arc : inout arc bus;
        out_arc : inout arc bus);
  END COMPONENT;
```

```
COMPONENT fork1
    generic (out_control : distrib;
    name : string;
    seed : integer;
    generation : frequency;
    debug_control : debug := none;
    report_freq : time := 10 fs);

  PORT (in_arc : inout arc bus;
    out_arc : inout arc_array (1 to
        number_out) bus);
END COMPONENT;
```

Queue Example Declarations

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

These declarations list the interface signals and generic parameters of the abstract data type components. The source1, queue1, and fork1 components are declared in this slide.

The ports are used to interconnect the elements of the queuing system. The generics list the static parameters used to configure particular component instantiations. For example, the queue length is defined as a generic. The values in the generic lists are default values in case specific values are not specified when the components are instantiated.

Socket Error

Connection to Remote Host timed out

Basic Module Format:

Architecture (cont.)

-- Notes Page --

This second part of the architecture description shows how the remaining three cycles of the handshake protocol (following the *placing* of a *token*) are implemented on both the data input port (if applicable) and the dependent output port (again, if applicable).

[UM93]

Three Module Example

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This example shows a simple three module model using ADEPT. The SOURCE provides *tokens* to the system at a rate of 1 every 5 ns. The FIXED_DELAY module introduces another 5 ns fixed delay. The time between new *tokens* is, therefore, 10 ns. Finally, the SINK module takes *tokens* out of the system.

[UM93]

Honeywell PML:

Bus Resolution Code

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

This slide shows a small portion of the Honeywell PML bus resolution function.

In this code, *s* is a vector of tokens passed to the BRF. If there are no drivers on the bus, the BRF returns an *init_token* (with a status of *idle*).

If there is only one token on the bus, the BRF returns that token.

If there is more than one token being driven, the BRF reads the protocol of the first token in the token vector *s*. Note that since all drivers on a bus must have the same protocol, only the first token needs to be checked. Multiple protocols on the same bus are supported through the use of the "CASE current_protocol" block and looping through all of the tokens in the vector.

To determine which token is to be returned, the BRF loops through all the tokens in the *s* token vector to determine the driver with the highest priority and state, and that token is returned.

[Honeywell95]

Honeywell PML Idle State

- Current_state of the bus is idle
- If a driver that is not idle is found, set current_state to that value

```
CASE current_state IS
  WHEN idle =>
    IF (s(i).state /=idle) THEN
      current_driver := i;
      current_state : = s(i).state;
    END IF;
```

Honeywell PML:

Idle State

-- Notes Page --

The value of *current_state* is defined to be the current state of the bus. *The current_state* value of the bus is used by the BRF to determine what token to return from the vector of token drivers for the bus.

If the state of the bus is *idle*, the BRF checks the vector of token drivers for a token with a state other than *idle*. If a non-*idle* token is found, that token is returned by the BRF.

[Honeywell95]

Honeywell PML Request State

- Current_state of the bus is request
- If multiple tokens are requesting, the one with higher priority wins
- Ack or busy tokens become current_state

```
WHEN request =>
    CASE s(i).state IS
      WHEN idle =>
      WHEN request =>
        IF (s(current_driver) <s(i)) THEN
          current_driver := i;
        END IF;
      WHEN ack | busy =>
        current_driver := i;
        current_state:= s(i).state;
  END CASE;
```


Honeywell PML:

Request State

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

If the current_state of the bus is *request*, execution of the BRF takes place in CASE statement. The "WHEN *idle* =>" line is left blank because a token can not be *idle* in this CASE block. Note that VHDL requires that all values in the range of an object be present in a CASE block (a "WHEN OTHERS =>" statement could have been used here instead to satisfy this requirement).

If the current token in the *s* vector is a *request*, the BRF checks this token's priority with that of the token that is the *current_driver* of the bus. If the new token has a higher priority, the new token becomes the *current_driver*. Note that the "<" operator has been overloaded to check for priority among tokens.

Finally, if the new token in the vector is an *ack* or *busy* token, this token is returned by the BRF and the *current_driver* and *current_state* variables are set accordingly.

[Honeywell95]

Honeywell PML Ack State

- Current_state of the bus is ack
- Multiple acks are not allowed
- Busy tokens set the current_state

```
WHEN ack =>
  CASE s(i).state IS
    WHEN idle =>
    WHEN request =>
    WHEN ack =>
    WHEN busy =>
      current_driver := i;
      current_driver := busy;
  END CASE;
```


Honeywell PML:

Ack State

-- Notes Page --

If the *current_state* of the bus is *ack*, the BRF checks the current token in the vector for the *busy* state. Because Honeywell does not allow multicast requests (multiple *request* tokens on the bus), there can be only one *ack* on the bus at a time.

[Honeywell95]

Honeywell PML Busy State

- Current_state of the bus is busy
- Multiple busys are allowed in lossy communication
- Busy token with highest priority wins the bus

```
WHEN busy =>
  CASE s(i).state IS
    WHEN idle =>
    WHEN request =>
    WHEN ack =>
    WHEN busy =>
      IF (s(current_driver) <s(i)) THEN
        current_driver := i;
      END IF;
  END CASE;
..
RETURN s(current_driver);
```


Honeywell PML:

Busy State

-- Notes Page --

Finally, if the *current_state* of the bus is *busy*, the BRF selects the *busy* token with the highest priority. Once again, the overloaded operator "<" is used to decide the higher priority.

[Honeywell95]

Honeywell PML:

Bus Interface Unit

IEEE VHDL LRM

VHDL HOME PAGE

-- Notes Page --

The Bus Interface Unit (BIU) is used to isolate a local bus from a global system bus.

[Honeywell95]

IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1993)

Sponsors

Design Automation Standards Committee of the IEEE Computer Society

and

Automatic Test Program Generation Subcommittee of IEEE Standards Coordinating Committee 20

Approved September 15, 1993
IEEE Standards Board
Approved April 14, 1994
American National Standards Institute

Abstract: This standard defines the VHSIC Hardware Description Language (VHDL). VHDL is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis , and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware. Its primary audience are the implementers of tools supporting the language and the advanced users of the language.

Keywords: Computer, computer languages, electronic systems, hardware, hardware design, VHDL

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-376-8

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. 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.

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 test, together with appropriate supporting comments.

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 all 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 technical 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 on standards and requests for interpretations should be addressed to:

```
Secretary, IEEE Standards Board
445 Hoes Lane
PO Box 1331
Piscataway, NJ  08855-1331
USA
```

IEEE Standards documents may involve the use of patented technology. Their approval by the Institute of Electrical and Electronics Engineers does not mean that using such technology for the purpose of conforming to such standards is authorized by the patent owner. It is the obligation of the user of such technology to obtain all necessary permissions.

Introduction

(This introduction is not a part of IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual.)

The VHSIC Hardware Description Language (VHDL) is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development,

verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware.

This document specifies IEEE Std 1076-1993, which is a revision of IEEE Std 1076-1987. The VHDL Analysis and Standardization Group (VASG) of the Computer Society of the IEEE started the development of IEEE Std 1076-1993 in June 1990. The VASG commissioned a Standardization Steering Committee to drive the standardization effort. The Steering Committee created standardization chapters in North America, Europe, and Asia-Pacific; administered the standardization guidelines of the IEEE; and staffed the volunteer positions in the various standardization chapters.

New capabilities in this version of the language include groups, shared variables, hierarchical pathnames, and a facility to include foreign models in a VHDL description. Some of the existing capabilities were extended or modified to facilitate initial and incremental creation of a design hierarchy. New shift/rotate operators were added to the language. The delay model of the language was modified to support pulse rejection. The syntactic consistency of the language was enhanced. Finally, resolutions of ambiguities and inconsistencies addressed by the Issue Screening and Analysis Committee (ISAC) of the VASG were incorporated into this revision of the language.

The VHDL 92 standardization effort consisted of five major phases: definition of VHDL 92 requirements, language design, language documentation, design validation, and balloting. The following working documents were developed during each phase of the standardization effort:

Requirements Definition:	VHDL 92 Requirements VHDL 92 Design Objectives Document
Language Design:	Language Change Specifications
Language Documentation:	Draft and Final Language Reference Manuals
Design Validation:	Validation Reports
Ballot Response Document:	Balloting

Numerous volunteers in North America, Europe, and Asia-Pacific contributed to development of VHDL 92. The Standardization Steering Committee consisted of the following:

Moe Shahdad	Steering Committee Chair
Stan Krolikoski	VASG Chair
Victor Berman	North-American Chapter Chair
Jean Mermet	European Chapter Chair
Kazuyuki Hirakawa	Asia-Pacific Chapter Chair
Jacques Rouillard	
Ron Waxman	
John Hillawi	
Andreas Hohl	

The following volunteers led the various working groups of the standardization effort:

Requirements Definition	Jacques Rouillard
Language Design	Doug Dunlop
Language Documentation	Paul Menchini
Design Validation	Alex Zamfirescu
Ballot Comment Resolution	Clive Charlwood

In addition, the following volunteers in the North America, European, and Asia-Pacific standardization chapters contributed to the VHDL 92 standardization effort by participating in the requirements gathering, requirements analysis, design review, documentation review, design validation, and balloting:

Mart Altmäe	Eric Gutt	Zainalabedin Navabi
Stephen A. Bailey	Andrew Guyler	Wolfgang Nebel
Daniel Barclay	William A. Hanna	Mary Lynne Nielsen
Jean-Michel Bergé	John Hines	Bill Paulsen
David Bernstein	Masaharu Imai	Hitomi Sato
Bill Billowitch	Kenichi Kanehara	Ken Scott
Dominique Borrione	Krishna Kumar	Sunder Singhani
Mark Brown	Oz Levia	Chuck Swart
Tedd Corman	Serge Maginot	Atushi Takahara
Alain Fonkoua	Erich Marschner	Cary Ussery
Rita Glover	Gabe Moretti	Eugenio Villar

IEEE Std 1076 is being maintained by the VASG. This group has been established to resolve issues that may arise with the language and to develop its future versions. The working documents of the VASG are available from the Computer Society Standards Secretariat, Computer Society of the IEEE, 1730 Massachusetts Ave. N.W., Washington, DC 20036, 1-202-371-0101, and also from the IEEE Standards Department, 445 Hoes Lane, Piscataway, NJ 08855, 1-800-678-IEEE. The working documents are not formally approved documents; however, they do reflect current status of the working group's direction.

As a result of the standardization activity leading to the development of IEEE Std 1076-1993, a number of working groups were formed to address areas that could not be adequately address within the scope of standardization:

Working Group	**Project Authorization Request (PAR) Number**
Shared Variables	1076.a
Analog Extensions	1076.1
Math Package	1076.2
Synthesis Package	1076.3
Timing Methodology	1076.4

Interested parties should contact the Chair of the Design Automation Standards Committee (DASC) to participate in these activities. *Development of IEEE Standard VHDL 1076-1987* IEEE Standard VHDL was developed through the work of the VASG, a working group within the Design Automation Standards Subcommittee (DASS) of the Design Automation Technical Committee (DATC) of the Computer Society of the IEEE. The work of the VASG was jointly sponsored by the DATC and by the Automatic Test Program Generation (ATPG) subcommittee of IEEE Standards Coordinating Committee 20 (SCC20). Larry Saunders was the Chair of the VASG; Ron Waxman was Chair of the DASS; Al Lowenstein was the Chair of the ATPG subcommittee. In the foreword to IEEE Std 1076-1987, Ronald Waxman (then Chair of DASS) and Larry Saunders (then Chair of VASG) acknowledged the efforts of Erich Marschner and Moe Shahdad as the principal designers of VHDL. They felt that the hard work and professionalism of the designers contributed significantly to the final result, and they wished the dedication of Marschner and Shahdad to be recognized. The creation of IEEE Std 1076-1987 began in February 1986 with the adoption of VHDL version 7.2 as the baseline language. In order to assist the voluntary standardization process of the IEEE, the Air Force Wright Aeronautical Laboratories contracted with CAD Language Systems Inc. (CLSI) to support the IEEE in the analysis of VHDL language issues, extension of the baseline language, and preparation of the draft and final definitions of the IEEE Std 1076-1987. This work was performed under contracts F33615-82-C-1716 and F33615-86-C-1050. The CLSI Project Manager for the IEEE standardization effort was Moe Shahdad, and the CLSI Technical Lead was Erich Marschner. The Air Force point of contact was John Hines, and Ron Waxman, Chair of the DASS, was the IEEE coordinator. Many individuals from many different organizations participated in the development of IEEE Std 1076-1987. In particular, the following people attended meetings of the VASG:

Dean Anderson	Ching Hsiao	Thomas Panfil

Kevin Anderson
Larry Anderson
Jim Armstrong
Lisa Asher
James Aylor
Jwahar Bammi
Peter Barck
Daniel Barclay
Dave Barton
Bill Beck
Victor Berman
Ken Caron
Hal Carter
Marc Casad
Moon Jung Chung
Patti Cochran
Dave Coelho
Doug Dunlop
Cathy Edwards
Thomas Elliot
Mike Endrizzi
Dave Evans
Deborah Frauenfelder
Mark Glewwe
Prabhu Goel
William Guzek
Jeff Haeffele
Charlie Haynes
John Hines
Mike Hirasuna
Ray Hookway
Paul Hubbard
Youm Huh
John Jensen
Bob Johnson
Susan Johnston
George Konstantino
Stan Krolikoski
Rick Lazansky
Jean Lester
Roger Lipsett
Shin-ming Liu
Al Lowenstein
Bruce Lundeby
Mark Macke
Robert Mackey
Erich Marschner
Paul Menchini
Lynn Meredith
Jean Mermet
Ellen Mickanin
Kieu Mien Le
Dwight Miller
Kent Moffat
Bob Morris
Jim Morris
Dan Nash
John Newkirk
Tim Noble
Ghulam Nurie
Leslie Orlidge
Ed Ott
Steve Piatz
Signe Post
Jean Pouilly
Bob Powell
Kim Rawlinson
Joel Rodriguez
Cary Sandvig
Larry Saunders
Lowell Savage
Tim Saxe
Dick Schlotfeldt
Peggy Schmidt
Ken Scott
Moe Shahdad
Arina Shainski
Alec Stanculescu
Stephen Sutherland
Tom Tempero
Jacques Tete
Tim Thorp
Tuan Tran
Stan Wagner
Rich Wallace
Karen Watkins
Ron Waxman
Isaiah White
Greg Winter
Craig Winton
Dan Youngbauer

1993 Development Record

The following persons were members of the balloting group that approved this standard for submission to the IEEE Standards Board:

William J. Abboud
Mostapha Aboulhamid
David Ackley
Guy Adam
Gordon Adshead
David G. Agnew
Gus Anderson
Kenneth R. Anderson
Walter Anheier
James R. Armstrong
Stephen A. Bailey
Pete Bakowski
Peter E. Barck
Daniel S. Barclay
Graham J. Barker
Akira Hasegawa
Greg Haynes
Frank Heile
John I. Hillawi
Robert Hillman
John Hines
Atsunobu Hiraiwa
Kazuyuki Hirakawa
Charles Homes
Paul W. Horstmann
Tamio Hoshino
Andy Huang
Stephen C. Hughes
Robert Stephen Hurley
Monique Hyvernaud
William R. Paulsen
Joseph Pick
Robert Piloty
Jean Pouilly
Jan Pukite
Sai V. Ramamoorthy
Edward P. Ratazzi
William E. Reeves
John P. Ries
Jean-Paul Rigault
Fred Rose
Charles W. Rosenthal
Jacques Rouillard
Paul Rowbottom
Susan Runowicz-Smith

John K. Bartholomew
Jean-Michel Berge
Victor Berman
David B. Bernstein
Dinesh Bettadapur
William D. Billowitch
Martin J. Bolton
Thomas H. Borgstrom
Dominique Borrione
Mark Brown
Patrick K. Bryant
Walter H. Burkhardt
Rosamaria Carbonell
Steven Carlson
Todd P. Carpenter
Harold W. Carter
Shir-Shen Chang
Clive R. Charlwood
Luc Claesen
Carl Cleaver
David A. Clough
David Coelho
John Colley
Frank Conforti
Tedd Corman
Robert A. Cottrell
Michael Crastes
Brian A. Dalio
Joseph P. Damore
Carlos Dangelo
Mark Davoren
Joanne DeGroat
Antonie deJager
Allen Dewey
Joseph P. Dorocak
Glenn E. Dukes
Dr. Michael Dukes
Douglas D. Dunlop
Nikil D. Dutt
Thomas D. Eberle
Rodney Farrow
Saverio Fazzari
Jacques P. Flandrois
Alain Fonkoua
Barbara Fredrick
Edmond Fumo
Benoit A. Gennart
Vassilios Gerousis
Alfred S. Gilman
Rita Glover
Yogesh Goel
Rich Goldman
Kenji Gotoh
Kazuhiko Iijima
Masaharu Imai
Nagisa Ishiura
Michel Israel
David Jakopac
Curtis Jensen
John E. Jensen
Susan M. Johnston
Hilary J. Kahn
Masaru Kakimoto
Takashi Kambe
Osamu Karatsu
Jake Karrfalt
Steve Kelum
Khozema Khambati
Choon B. Kim
Eskil Kjelkerud
Paul Knese
Tokinori Kozawa
Albert J. Kreutzer
Stanley J. Krolikoski
Howard K. Lane
Kin Sing Lau
Oz Levia
Paul A. Lewis
Stephen Lim
Alfred Lowenstein
Martin J. Lynch
Don MacMillen
Serge Maginot
Leon I. Maissel
Erich Marschner
Peter Marwedel
Gayle Matysek
Pankaj Mayor
George A. Mazoko
Robert L. McGarvey
Sean McGoogan
William S. McKinney
Paul J. Menchini
Jean Mermet
Dwight L. Miller
John T. Montague
Gabe Moretti
David S. Morris
Wolfgang Mueller
Pradipto Mukherjee
Jack Mullins
Shinichi Murai
Satish Nagarajan
Jayant L. Nagda
Hiroshi Nakamura
Michael P. Nassif
William E. Russell, Jr.
Michael Ryba
Ashraf M. Salem
Hitomi Sato
Larry F. Saunders
Paul Scheidt
Paul W. Schlie
Kenneth E. Scott
Jorge Seidel
Francesco Sforza
Moe Shahdad
Ravi Shankar
Takao Shinsha
Isao Shirakawa
Lee A. Shombert
Supreet Singh
Sunder Singhani
John Sissler
Djahida Smati
J. W. Smith
Dennis Soderberg
Jay R. Southard
Joseph J. Stanco
Alec G. Stanculescu
Balsha R. Stanisic
Charles Swart
Atsushi Takahara
Kiyotaka Teranishi
Jacques Tete
Jose A. Torres
Carl W. Traber
S. Tracey
Andy S. Tsay
Jean Pierre Tual
Cary Ussery
Radha Vaidyanathan
Sai K. Vedantam
Kerry Veenstra
James H. Vellenga
Ranganadha R. Vemuri
Venkat V. Venkataraman
Eugenio Villar
Malcolm Wallace
Xinning Wang
Karen E. Watkins
Ronald Waxman
J. Richard Weger
Ron Werner
Gregory Whitcomb
Francis Wiest
Paul S. Williams
John C. Willis
G. Winter

James Graves
Arnold Greenspan
Brent L. Gregory
Brian Griffin
Paul-Marie Grojean
Steve Grout
Laurence T. Groves
Andrew Guyler
Jeffrey J. Haeffele
Claes L. Hammar
William A. Hanna
James P. Hanna
John W. Harris
Damon C. Hart
Zainalabedin Navabi
Sivaram Nayudu
Dr. Wolfgang H. Nebel
Richard E. Neese
Gordon Newell
Meyer Elias Nigri
Ryo Nomura
Nancy Nugent
John W. O'Leary
Tetsuya Okabe
Vincent Olive
Yoichi Onishi
Catherine Ozenfant
Curtis Parks
James L. Wong
Akihiko Yamada
Hiroto Yasuura
Ping Yeung
Joseph M. Youmans
Will W. Young
Simon Young
Tonny Yu
Tetsuo Yutani
Alex Zamfirescu
Guoqing Zhang
Reinhard Zippelius

When the IEEE Standards Board approved this standard on September 15, 1993, it had the following membership:

Wallace S. Read, *Chair* **Donald C. Loughry**, *Vice Chair*
Andrew G. Salem, *Secretary*

Gilles A. Baril
José A. Berrios de la Paz
Clyde R. Camp
Donald C. Fleckenstein
Jay Forster*
David F. Franklin
Ramiro Garcia
Donald N. Heirman
Jim Isaak
Ben C. Johnson
Walter J. Karplus
Lorraine C. Kevra
E. G. "Al" Kiener
Ivor N. Knight
Joseph L. Koepfinger*
D. N. "Jim" Logothetis
Don T. Michael*
Marco W. Migliaro
L. John Rankine
Arthur K. Reilly
Ronald H. Reimer
Gary S. Robinson
Leonard L. Tripp
Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
James Beall
Richard B. Engelman
David E. Soffrin
Stanley I. Warshaw

Mary Lynne Nielsen
IEEE Standards Project Editor

This standard has been adopted for Federal Government use.

Details concerning its use within the Federal Government are contained in Federal Information Processing Standards Publication 172-1, VHSIC Hardware Description Language (VHDL). For a complete list of publications available in

the Federal Information Processing Standards series, write to the Standards Processing Coordinator, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899.