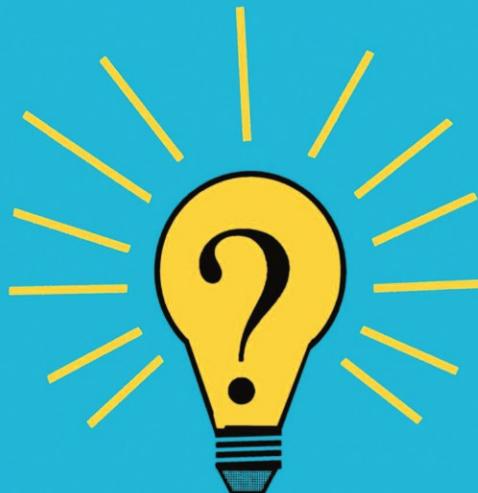


# VHDL

## Answers to Frequently Asked Questions

## Second Edition



**Ben Cohen**



---

---

**VHDL ANSWERS TO FREQUENTLY  
ASKED QUESTIONS**  
**Second Edition**

---

VHDL ANSWERS TO FREQUENTLY  
ASKED QUESTIONS  
Second Edition

by

**Ben Cohen**



SPRINGER SCIENCE+BUSINESS MEDIA, LLC

### **Library of Congress Cataloging-in-Publication Data**

A C.I.P. Catalogue record for this book is available  
from the Library of Congress.

**ISBN 978-1-4613-7581-4      ISBN 978-1-4615-5641-1 (eBook)**  
**DOI 10.1007/978-1-4615-5641-1**

---

**Additional material to this book can be downloaded from <http://extras.springer.com>.**

(1) Reprinted from IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual,  
Copyright© 1994 by the Institute of Electrical and Electronics Engineers, Inc.

The IEEE disclaims any responsibility or liability resulting from the placement (in hard copy or electronic format) and use in this publication of the following documents and packages:

- IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual, Copyright© 1994 by the Institute of Electrical and Electronics Engineers, Inc.
- Package STANDARD and package TEXTIO from the IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual.
- IEEE Std 1164-1993 IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std Logic 1164), Copyright© 1993 by the Institute of Electrical and Electronics Engineers, Inc.
- IEEE Draft Standard P1076.3, Version: 2.4, Dated 12 April 1995, Copyright© 1995 by the Institute of Electrical and Electronics Engineers, Inc.. This is an unapproved draft of a proposed IEEE standard, subject to change. Use of information contained in the unapproved draft is at your own risk.

Information is reprinted with the permission of the IEEE.

**Copyright © 1998 Springer Science+Business Media New York  
Originally published by Kluwer Academic Publishers, New York in 1998  
Softcover reprint of the hardcover 2nd edition 1998**

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photo-copying, recording, or otherwise, without the prior written permission of the publisher, Springer Science+Business Media, LLC.

*Printed on acid-free paper.*

I dedicate this book to my wife, Gloria Jean.  
Her creative and organizational contributions  
proved to be invaluable in the design of the  
cover, and the format and title of the book.  
Her support throughout this project is deeply  
appreciated.

# CONTENTS

<b>PREFACE</b>	<b>xv</b>
<b>ABOUT THE DISK</b>	<b>xix</b>
<b>NOTATION CONVENTIONS</b>	
<b>Symbols</b>	<b>xxv</b>
<b>Syntactic Description</b>	<b>xxvi</b>
<b>ACKNOWLEDGMENTS</b>	<b>xxvii</b>
<b>ABOUT THE AUTHOR</b>	<b>xxviii</b>
<b>1. LANGUAGE ELEMENTS</b>	<b>1</b>
<b>1.1 WHY VHDL FOR DIGITAL DESIGNS</b>	<b>4</b>
<b>1.2 SALIENT POINTS OF CONCURRENT STATEMENTS</b>	<b>6</b>
1.2.1 Process	6
1.2.2 Concurrent Signal Assignment Statements	7
1.2.3 Component Instantiation	8
1.2.4 Concurrent Procedure	8
1.2.5 Generate	9
1.2.6 Concurrent Assertion	9
1.2.7 Block statement	10
<b>1.3 GUARDED SIGNAL ASSIGNMENTS</b>	<b>11</b>
<b>1.4 SIGNALS AND PORTS</b>	<b>17</b>
1.4.1 Disconnecting a Signal	17
1.4.2 Difference between <i>inout</i> and <i>buffer</i> ports	17
1.4.3 Port Association Rules – the Open	20
1.4.4 Type Conversion in Port Associations	22
<b>1.5 CONFIGURATIONS</b>	<b>24</b>
1.5.1 Configuration Requirements	24
1.5.2 Configuration Specification/Declaration	25
1.5.2.1 Configuration Specification	25
1.5.2.2 Configuration Declaration	28
1.5.2.3 Binding with configured components	31
1.5.2.4 Removing a Component from an Architecture through Configuration	31
1.5.3 Configuration Statement for a Specific Generic	32
1.5.4 CONFIGURATION OF GENERATE STATEMENTS	33
<b>1.6 ARITHMETIC ISSUES AND OPERATORS</b>	<b>38</b>
1.6.1 Defining 2'S Complement Value -(2**31) in VHDL	38
1.6.2 Value Casting	39
1.6.3 REM/MOD Difference	40
1.6.4 Non-Associative Operators	41

1.6.5 Analysis Error with the “&” Operator	42
1.6.6 Specifying a Multi-Line String.	43
<b>1.7 PACKAGE STD_LOGIC_1164</b>	<b>43</b>
1.7.1 Differences Std_Logic_1164'1987 and '1993	43
1.7.2 Accessing Multiple Fields of Std_Logic_Vector	44
1.7.3 Aliases and Constants	44
1.7.4 Enumeration Type	46
1.7.4.1 Separate conversion functions.	46
1.7.4.2 Use of Integer Subtypes	47
1.7.4.3 Use of 'val' attributes	47
1.7.5 Testbench for Demonstration Models	48
<b>1.8 RANGE CONSTRAINT IN TYPE DEFINITION</b>	<b>50</b>
<b>1.9 SHARED VARIABLES</b>	<b>50</b>
<b>2. ARRAYS</b>	<b>55</b>
<b>2.1 ARRAY STRUCTURE REPRESENTATIONS</b>	<b>56</b>
2.1.1 One Dimensional Arrays	56
2.1.2 Multi-Dimensional Arrays	56
<b>2.2 ARRAYS -- LEGAL OPERATIONS</b>	<b>58</b>
2.2.1 Overloaded Operators on Arrays	60
<b>2.3 ARRAY SLICES AND RANGES</b>	<b>65</b>
<b>2.4 ARRAY INITIALIZATION</b>	<b>67</b>
<b>2.5 CONSTANT ARRAYS IN CASE</b>	<b>70</b>
<b>2.6 CONSTRAINED AND UNCONSTRAINED ARRAYS</b>	<b>71</b>
2.6.1 Constraining Methods	72
2.6.2 Allowed Constrained and Unconstrained Objects	73
<b>2.7 MAPPING ARRAYS OF DIFFERENT SIZES</b>	<b>74</b>
<b>2.8 UNCONSTRAINED AGGREGATE WITH "OTHERS "</b>	<b>75</b>
<b>2.9 ILLEGAL ARRAY TYPES</b>	<b>78</b>
2.9.1 UNCONSTRAINED ARRAY OF AN UNCONSTRAINED ARRAY	78
<b>3. DRIVERS</b>	<b>81</b>
<b>3.1 MULTIPLE DRIVERS - CASE 1</b>	<b>82</b>
3.1.1 Std_Logic_Vector Solution	83
3.1.2 Separate Signals Solution	83
3.1.3 Atomicity	84

Table of Contents	ix
<b>3.2 MULTIPLE DRIVERS - CASE 2</b>	91
<b>3.3 MULTIPLE DRIVERS ERROR – CASE 3</b>	92
<b>3.4 MULTIPLE DRIVERS ERROR – COMPONENT</b>	93
<b>3.5 CODING STYLE FOR DETECTION OF MULTIPLE DRIVERS</b>	95
<b>4. SUBPROGRAMS</b>	<b>99</b>
<b>4.1 SIDE EFFECTS FROM A PROCEDURE</b>	100
4.1.1 Language View	100
4.1.2 STYLE VIEW	103
<b>4.2 GARBAGE COLLECTION OF DYNAMICALLY CREATED OBJECTS</b>	106
<b>4.3 ACCEPTABLE TYPES IN PARAMETER LISTS FOR FUNCTION CALLS</b>	110
<b>4.4 FILES DECLARATIONS IN PROCEDURES</b>	111
<b>4.5 MULTIPLE ACCESSES OF SAME FILE</b>	113
<b>4.6 FILE ARRAY</b>	114
<b>4.7 CONVERSION FUNCTION FROM INTEGER TO TIME</b>	116
<b>4.8 NORMALIZATION IN SUBPROGRAMS</b>	118
<b>5. PACKAGES</b>	<b>121</b>
<b>5.1 CONVERTING TYPED OBJECTS TO STRINGS</b>	122
<b>5.2 PRINTING OBJECTS FROM VHDL</b>	126
5.2.1 Writing to One file from Multiple Processes	129
<b>5.3 MULTIPLE INPUT SIGNATURE REGISTER (MISR)</b>	130
<b>5.4 DESIGN OF A LINEAR FEEDBACK SHIFT REGISTER (LFSR)</b>	132
<b>5.5 RANDOM NUMBER GENERATION</b>	137
<b>5.6 DEFERRED CONSTANT IN PACKAGE DECLARATION</b>	138
<b>5.7 COMPLEX NUMBERS AND OVERLOADED OPERATORS</b>	138
<b>5.8 IEEE STANDARDS</b>	143
<b>6. MODELS</b>	<b>145</b>

<b>6.1 LARGE RAM MODEL FOR SIMULATION.</b>	<b>146</b>
6.1.1 Traditional Memory Modeling	146
6.1.2 Efficiency Memory Modeling	149
6.1.3 Fixed Array Caching	149
6.1.4 Fixed Page Caching	152
6.1.5 Dynamic Page Caching	154
6.1.6 Disk paging with swap files	156
6.1.7 C Language Interface	157
6.1.8 RAM Testbench and Configurations	157
<b>6.2 ZERO OHM RESISTOR (WIRE, BRIDGE) MODEL</b>	<b>159</b>
6.2.1 Zero Ohm Resistor Model Restrictions	162
6.2.2 Architecture Variation Method for Resistor	163
<b>6.3 ERROR INJECTOR MODEL</b>	<b>163</b>
<b>6.4 TRANSFER GATE (SWITCH)</b>	<b>178</b>
<b>7. SYNTHESIS</b>	<b>183</b>
<b>    7.1 SUPPORTED/UNSUPPORTED CONSTRUCTS FOR SYNTHESIS</b>	<b>184</b>
<b>    7.2 SYNTHESIS SENSITIVITY RULES</b>	<b>186</b>
<b>    7.3 LATCH/REGISTER/COMBINATIONAL LOGIC</b>	<b>186</b>
<b>    7.4 LATCH INFERRANCE IN FUNCTIONS</b>	<b>193</b>
<b>    7.5 VARIABLE INITIALIZATION AND LIFETIME</b>	<b>193</b>
7.5.1 Variable Initialization in Processes	193
7.5.2 Variable Initialization for Subprograms (Called from Processes)	195
<b>    7.6 WAIT STATEMENT</b>	<b>195</b>
<b>    7.7 DEFINING SHIFT REGISTERS IN SYNTHESIS</b>	<b>197</b>
7.7.1 Resolution -- Code for Shift Register	197
<b>    7.8 REGISTER FILE</b>	<b>199</b>
7.8.1 Register File – with signals	199
7.8.2 Register File – with variables	200
<b>    7.9 MULTIPLEXER MODEL</b>	<b>202</b>
<b>    7.10 DEMULTIPLEXER MODEL</b>	<b>203</b>
<b>    7.11 BARREL SHIFTER</b>	<b>205</b>
<b>    7.12 USE OF "DON'T CARE" IN CASE STATEMENT</b>	<b>207</b>
7.12.1 Case Statement with No "Don't Care"	208
7.12.2 IF Statement	210

<b>7.12.3 Loop Statement</b>	210
<b>7.12.4 IEEE Std_Synth</b>	211
 <b>7.13 PARAMETERIZED PRIORITY ENCODER</b>	 212
<b>7.13.1 Straight Encoding Priority Encoder</b>	213
<b>7.13.2 Two level encoding</b>	215
 <b>7.14 GENERATING A SYNCHRONOUS PRECHARGE</b>	 220
 <b>7.15 TECHNOLOGY AND VHDL CODE DESIGN</b>	 223
<b>7.15.1 Basic architecture of a logic cell.</b>	223
<b>7.15.2 Synthesis tool technology</b>	226
<b>7.15.3 Supporting component libraries</b>	226
 <b>7.16 SYNTHESIZING TRI-STATES</b>	 226
<b>7.16.1 Latches in Tri-State Controls</b>	227
 <b>7.17 SUBELEMENT ASSOCIATION</b>	 230
<b>7.17.1 Subelement Association for Ports</b>	231
 <b>7.18 FINITE STATE MACHINE (FSM)</b>	 232
 <b>7.19 ONE-HOT ENCODING</b>	 233
 <b>7.20 INSTANTIATING SYNOPSYS® DESIGNWARE COMPONENTS</b>	 234
 <b>7.21 RESOURCE SHARING</b>	 235
 <b>7.22 APPLYING DIGITAL DESIGN EXPERIENCES</b>	 236
 <b>7.23 ADDRESS RANGE IDENTIFICATION VIA INFERRED COMPARATOR</b>	 238
 <b>7.24 PORT MAPPING TO GROUND OR VCC</b>	 239
 <b>7.25 BIT REVERSAL</b>	 240
 <b>7.26 HOW TO DESIGN A TIMER IN VHDL</b>	 241
 <b>7.27 SPECIFYING A MULTIPLIER</b>	 242
 <b>7.28 BEHAVIORAL SYNTHESIS</b>	 243
 <b>8. DESIGN VERIFICATION AND TESTBENCH</b>	 245
 <b>8.1 VERIFICATION PROCESSES</b>	 246
 <b>8.2 FUNCTIONAL VERIFICATION</b>	 247
 <b>8.3 REGRESSION TESTS</b>	 250
<b>8.3.1 File Compare Method</b>	250

8.3.2 Design Verifier Method	251
8.3.3 MISR Method	251
8.3.4 Formal Verification Method	252
<b>8.4 FORMAL VERIFICATION</b>	<b>252</b>
8.4.1 How Formal Verification Works	252
8.4.2 How Formal Verification Fits in the Design Cycle	253
8.4.3 Formal Verification Advantages	253
<b>8.5 BUS FUNCTIONAL MODEL (BFM) MODELING</b>	<b>254</b>
<b>8.6 APPLICATION OF MISR, RANDOM, LFSR PACKAGES FOR AUTO-REGRESSION</b>	<b>257</b>
<b>8.7 STRENGTH STRIPPER</b>	<b>267</b>
<b>9. POTPOURRI</b>	<b>269</b>
<b>9.1 METHODS TO ENHANCE SIMULATION SPEED</b>	<b>270</b>
9.1.1 Signals	270
9.1.1.1 Use Signals for Inter-Process Communication Only	271
9.1.1.2 Avoid Signal Reassignment for Same Value	271
9.1.1.3 Avoid Individual Assignment of Individual Elements of a Composite	271
9.1.1.4 Reduce Number of Signals	272
9.1.1.5 Avoid resolved signals	272
9.1.2 Concurrent Statements	273
9.1.2.1 Number of Processes/Concurrent Statements	273
9.1.2.2 Concurrent Signal Assignment, Sensitivity list or Wait Statements <sup>[6]</sup>	273
9.1.2.3 Processes with Multiple Clocks	275
9.1.2.4 Disabling Non-Necessary Concurrent Statements	275
9.1.2.5 Avoid Repeated Code or Function Calls	277
9.1.2.6 Avoid Unnecessary Processes	278
9.1.3 Types	278
9.1.4 Constants	279
9.1.5 Files	280
9.1.6 Subprograms	284
9.1.7 Memory	284
9.1.8 Packages	285
9.1.9 VITAL	285
<b>9.2 ACCESSING SIGNALS INTERNAL TO COMPONENTS</b>	<b>285</b>
9.2.1 Global Signal Method	285
9.2.2 PORT Method	289
<b>9.3 TRANSFERRING A LINE ONTO A SIGNAL</b>	<b>291</b>
<b>9.4 TYPE DECLARATION IN MULTIPLE PACKAGES</b>	<b>293</b>
<b>9.5 INTERNET - FREQUENTLY ASKED QUESTIONS</b>	<b>294</b>

<b>9.6 VHDL TEXT EDITOR?</b>	<b>294</b>
<b>9.7 VITAL</b>	<b>295</b>
9.7.1 VITAL Features	295
<b>9.8 BEHAVIORAL MODELING</b>	<b>296</b>
<b>9.9 FINAL VHDL EXAM</b>	<b>297</b>
9.9.1 Design Units	297
9.9.2 Compilation Order	298
9.9.3 VHDL Types	300
9.9.4 Attributes	302
9.9.5 “WAIT” Statement	302
9.9.6 Control Structure	303
9.9.7 Signals versus Variables	304
9.9.8 Operator Overloading	306
9.9.9 Model	306
9.9.10 Concurrent Statements	309
9.9.11 Drivers and Resolution Functions	309
9.9.12 Subprogram	311
<b>10. DESIGN FOR REUSE</b>	<b>313</b>
<b>10.1 DESIGN PROCESSES FOR REUSABILITY</b>	<b>314</b>
<b>10.2 PARAMETERIZED, REUSABLE AND READABLE CODE</b>	<b>316</b>
10.2.1 Notation	316
10.2.2 Path Definition	317
10.2.3 “Accordion” Parameterization	318
10.2.4 “Control” Parameterization	319
10.2.5 <i>Block</i> Statements for Partition Separations	323
10.2.6 <i>Generate</i> Statement for Multiple Instantiations of Components	328
10.2.7 Subprograms	328
10.2.8 Use of VHDL Attributes	328
10.2.9 Core Models	329
10.2.10 Bus Functional Model (BFM) and Testbench Architecture	330
<b>10.3 DOCUMENTATION OF VHDL DESIGNS</b>	<b>331</b>
10.3.1 Summary	331
10.3.2 Applicable Documents/References	331
10.3.3 Definitions	331
10.3.4 Design Team	331
10.3.5 Requirement Versus Design Matrix	332
10.3.6 Design Description	332
10.3.6.1 Interface Definitions	332
10.3.6.2 Block Diagrams and Registers	333
10.3.6.3 State Diagrams	333
10.3.6.4 Design Rationale	333
10.3.6.5 Operation Modes	333
10.3.6.6 Instruction Set Architecture ( <i>ISA</i> )	334

10.3.6.7 Software Interface (Hardware/Firmware) Model	334
10.3.6.8 Design Restrictions and Ambiguities	334
10.3.6.9 Model Scalability	334
10.3.6.10 Design Testability	334
10.3.6.11 Hardware Initialization	335
10.3.6.12 Electrical Specification	335
10.3.6.13 Timing specifications	335
10.3.6.14 Design Files and compilation order	335
10.3.6.15 Application notes	335
10.3.7 Synthesis	336
10.3.7.1 Synthesis tools	336
10.3.7.2 Synthesis Library	336
10.3.7.3 Design Constraints	336
10.3.7.4 Compilation Scripts	336
10.3.7.5 Performance	336
10.3.8 Technology Mapping and Routing	336
10.3.9 Verification	337
10.3.9.1 Validation plan [9].	337
10.3.9.2 Testbench and Verifier.	338
10.3.10 Simulation	338
10.3.10.1 Platforms	338
10.3.10.2 Design Files and compilation order	339
10.3.10.3 Simulation Results	339
10.3.11 Backup Media	339
10.3.12 Recommendations	339
10.3.13 Index	339
<b>APPENDIX A: VHDL'93 AND VHDL'87 SYNTAX SUMMARY</b>	<b>341</b>
<b>APPENDIX B: PACKAGE STANDARD</b>	<b>351</b>
<b>APPENDIX C: PACKAGE TEXTIO</b>	<b>353</b>
<b>APPENDIX D: PACKAGE STD_LOGIC_1164</b>	<b>355</b>
<b>APPENDIX E: PACKAGE STD_LOGIC_ARITH</b>	<b>359</b>
<b>APPENDIX F: VHDL PREDEFINED ATTRIBUTES</b>	<b>365</b>
<b>BIBLIOGRAPHY</b>	<b>369</b>
<b>INDEX</b>	<b>371</b>

# PREFACE

*VHDL Answers to Frequently asked Questions* is a follow-up to the author's book *VHDL Coding Styles and Methodologies* (ISBN 0-7923-9598-0). On completion of his first book, the author continued teaching VHDL and actively participated in the *comp.lang.vhdl* newsgroup. During his experiences, he was enlightened by the many interesting issues and questions relating to VHDL and synthesis. These pertained to: misinterpretations in the use of the language; methods for writing error free, and simulation efficient, code for testbench designs and for synthesis; and general principles and guidelines for design verification. As a result of this wealth of public knowledge contributed by a large VHDL community, the author decided to act as a facilitator of this information by collecting different classes of VHDL issues, and by elaborating on these topics through complete simulatable examples.

This book is intended for those who are seeking an enhanced proficiency in VHDL. Its target audience includes:

1. **Engineers.** The book addresses a set of problems commonly experienced by real users of VHDL. It provides practical explanations to the questions, and suggests practical solutions to the raised issues. It also includes packages of common utilities that are useful in the generation of debug code and testbench designs. These packages include conversions to strings (the IMAGE package), generation of Linear Feedback Shift Registers (LFSR), Multiple Input Shift Register (MISR), and random number generators. It also defines models of useful components including large, but memory-on-demand Random Access Memory (RAM), zero ohm resistor, and error injector. It includes synthesis guidelines and rules, and demonstrates these concepts through the modeling of several primitives. It provides a discussion and guidelines on design verification techniques, and describes methods to enhance simulation speed. A methodology for testbench designs is also defined. All VHDL code described in the book is on a companion 3.5" PC disk.
2. **College students.** This book enhances a student's comprehension of the language by addressing the user's concerns. It furnishes the students with useful examples that can be compiled and simulated.

This book differs from other VHDL books in the following respects:

1. Emphasizes real VHDL problems, rather than philosophical or introductory type of information.
2. Emphasizes application of VHDL for synthesis.
3. Uses complete examples to demonstrate problems and solutions.

4. Provides a disk that includes all the book examples and other useful reference VHDL material.
5. Uses easy to remember symbology notation to emphasize language rules, good and poor methodology and coding styles.
6. Identifies obsolete VHDL constructs that must be avoided.
7. Identifies synthesizable/non-synthesizable structures.
8. Uses a question and answer format to clarify and emphasize the concerns of VHDL users.

The book is organized in nine sections, starting with language related issues and progressing to real applications of VHDL. These sections include:

1. **LANGUAGE ELEMENTS.** Addressed are the issues related to the basic language elements of VHDL, including the salient points of concurrent statements, configurations, ports, arithmetic issues, and package Std\_Logic\_1164.
2. **ARRAYS.** Addressed are array operations, array initialization, use of constrained and unconstrained arrays, and mapping of arrays of different sizes. The application of arrays in synthesis is also addressed
3. **DRIVERS.** Errors in the creation of unexpected multiple drivers are examined.
4. **SUBPROGRAMS.** The issues of side effects, garbage collection, files, conversion functions, and normalization of parameters are discussed.
5. **PACKAGES.** Useful packages are examined. These include conversion of various types to binary, integer and hex strings; Multiple Input Signature Register (MISR); and Linear Feedback Shift Register (LFSR) definitions and applications; and random number generation and applications.
6. **MODELS.** Several classes of VHDL concepts are demonstrated through the definition of models. These models demonstrate various methods to implement a RAM, including a memory efficient model using a linked list and paging approach. It also includes a model of a zero ohm resistor and its applications. A model of an error injector and its applications is also provided.
7. **SYNTHESIS.** This section identifies the supported and unsupported constructs for synthesis. It also identifies a class of synthesis problems and solutions using a variety of primitive models. It identifies when synthesis interprets certain objects as registers, latches, or combinational logic. Synthesis of tri-states and design optimization is also discussed.
8. **DESIGN VERIFICATION AND TESTBENCH.** Various design verification approaches are examined, including: functional verification, auto-regression, and formal verification. Design method for testbench modeling is extensively addressed.
9. **POTPOURRI.** This section is a collection of topics that did not fit into a specific set of classification. It particularly emphasizes the methods to enhance simulation speed.

10. **DESIGN FOR REUSE.** This section discusses the disciplines in design processes, VHDL coding styles and documentation to achieve optimum design reuse. The disciplines in design processes address the issues of integrating pre-designed models and of entering into the reuse data base newly defined items. The VHDL Coding style discipline addresses the issues of partitioning, parameterization, and model design for readability, flexibility and reuse. The documentation discipline addresses the issues of good documentation to enable design reuse and better communication.

# About The Disk

This disk contains all of the VHDL code presented in the chapters. Table 1 summarizes the contents of the disk.

**Table 1. Contents of Enclosed Disk**

Directory Name	File Name	Figure #	Description
root	disclaim.txt	---	Disclaimer
element	alias1.vhd	---	Aliases in an entity
element	alist_ea.vhd	1.5.4-1	Port association list
element	alisttb.vhd	1.5.4-1	Examples of port association conversion rules, VHDL'87
element	alisttb2.vhd	1.5.4-2	Examples of port association conversion rules, VHDL'93
element	attrible.vhd	1.8.4.3	Use of 'val attribute to enhance code readability
element	block.vhd	1.3.7-1	Block example in synthesis
element	blockh.vhd	1.3.7-2	Block example with port header
element	buffer.vhd	1.5.2-4	Alternative use of buffer ports with <i>out</i> ports
element	buffer2.vhd	1.5.2-1	Component with buffer port
element	bufftop2.vhd	1.5.2-2	Instantiation of a component with <i>buffer</i> error
element	bufftop3.vhd	1.5.2-3	Instantiation of a component with <i>buffer</i> error
element	cfgdecl.vhd	1.6.2.2-1	Architecture and configuration declaration example
element	cfgspec.vhd	1.6.2.1	Configuration specification example
element	cntlr_c.vhd	1.8.5-2	Configuration declaration for controller model
element	cntlr_e.vhd	1.8.3-1	Aliases to enhance readability
element	cntlrb.vhd	1.8.5-1	Testbench for controller model
element	concat.vhd	1.7.5-2	Error in case expression
element	concat2.vhd	---	Same as concat.vhd, but error code commented out
element	confused.vhd	1.8.1	Differences in <i>Std_Logic_1164'87</i> and <i>Std_Logic_1164'93</i>
element	constant.vhd	1.8.3-2	Aliases and constants to enhance readability
element	convert.vhd	---	non-synthesizable version of <i>consynt.vhd</i> using aliasing
element	convn.vhd	1.7.2	Examples of VHDL type conversions
element	convsynt.vhd	1.8.4.1-1	Conversion function to enhance code readability
element	count_ea.vhd	1.7.3-2	<i>Mod</i> operator for a count-down counter
element	cproc.vhd	1.3.4	Concurrent procedure with side effect
element	deferd_c.vhd	1.6.2.4	Configuration to remove a component from an architecture
element	filter2.vhd	1.6.3	Setting generics with configuration declarations
element	genrat_c.vhd	1.6.4-1	Configuration of generate statement
element	hierarch.vhd	1.6.2.2-3	Binding of components of a hierarchical design with a configuration declaration
element	left2r.vhd	1.7.4	Non-associative operators – errors
element	open_ea.vhd	1.5.3	Port association examples
element	process.vhd	1.3.1	<i>Wait</i> statements in process
element	shared.vhd	1.10	Two processes exclusively accessing a shared resource
element	ssasgmt.vhd	1.3.2-1	Selected signal assignment
element	struct3.vhd	1.6.2.3	Binding with configured component
element	T4guard.vhd	1.4-4	Application of guards

Directory Name	File Name	Figure #	Description
element	Tguard.vhd	1.4-1	Application of guards – testbench
element	topcfg.vhd	1.6.4-1	Configuration of a hierarchical design with <i>generate</i>
element	static.vhd	1.9	Subtype definition
element	wave.vhd	1.3.2-2	Waveform generation
arrays	anony_ea.vhd	2.9	Illegal anonymous array
arrays	array1.vhd	2.2	Implicit operators on arrays
arrays	ary.vhd	2.6	Unconstrained and constrained array restrictions
arrays	aryinit.vhd	2.4-1	Array initialization error
arrays	casesct.vhd	2.5-2	Case statement with locally static expression
arrays	casesct2.vhd	2.5-1	Case statement with error
arrays	generic.vhd	2.4-2	Illegal referencing of an identifier
arrays	multidm.vhd	2.7	Mapping of arrays of different sizes
arrays	signed.vhd	2.2.1-2	Overloaded "+" operator on signed and unsigned objects
arrays	test.vhd	2.9.1-1	Constraining array size based on size of other types, generics, and deferred constants
arrays	tmspec.vhd	2.1.2-2	Time parameters with two-dimensional arrays
arrays	uncons.vhd	2.8-1	Unconstrained aggregate with <i>others</i> -- errors
arrays	unconsok.vhd	2.8-2	Constrained aggregate with <i>others</i>
arrays	SynopOk.vhd	2.8-3	Assignment of a Slice with Synopsys ()
drivers	atmdrv5.vhd	3.1.3-2	Process sensitivity on resolved signal
drivers	atombit5.vhd	---	Process sensitivity on unresolved signal
drivers	atomic_pb.vhd	3.1.3-1	Package demonstrating resolved array type
drivers	atomic.vhd	---	Process sensitivity on resolved and unresolved signal
drivers	atomic2.vhd	---	Process sensitivity on resolved and unresolved signal
drivers	atomic3.vhd	---	Process sensitivity on resolved and unresolved signal
drivers	atomic5.vhd	3.1.3-4	Sensitivity on processes with resolved signal
drivers	atomstd5.vhd	---	Process sensitivity on Std_Logic_Vector
drivers	bug.vhd	3.3-1	Multiple drivers error
drivers	bugfix.vhd	3.3-2	fix of bug for file <i>bug.vhd</i>
drivers	drvstd5.vhd	---	Drivers and sensitivity on Std_Logic_Vector
drivers	drvbit.vhd	3.1	Problem with multiple drivers
drivers	drvstd.vhd	3.1.1	Problem with drivers
drivers	drvstd2.vhd	3.1.2	Separate signals resolution to avoid multiple drivers
drivers	recre.vhd	3.2-1	Multiple drivers – errors
drivers	recrdok.vhd	3.2-2	Multiple drivers resolution with static indices
drivers	test.vhd	3.4-1	Multiple drivers – components
drivers	testok.vhd	3.3-4	Resolution of multiple drivers for file <i>test.vhd</i>
drivers	uresolve2.vhd	3.5-2	Levels of Complexity when using Unresolved signal and Arithmetic Operations
drivers	uresolve.vhd	3.5-1	Instantiating a Component with Unresolved Ports
subprgm	access.vhd	4.2-2	Aggravated case of a leaky subprogram
subprgm	accessok.vhd	4.2-3	Non-Leaky subprogram
subprgm	bitrfnct.vhd	4.8-1, -2	Normalized and Non-normalized variables in subprogram
subprgm	datain.txt	---	support file
subprgm	drvproc.vhd	4.1.2-2	Client/server approach to avoid side effects
subprgm	f_ary87.vhd	4.6-1	Indexed array model (VHDL'87)
subprgm	\wrln87.vhd	4.2-4	Code to Test for a Memory Leak Condition
subprgm	wrifx87.vhd		Fix for readline memory leak
subprgm	fcopy87.vhd	4.4-1	Procedure to copy a file (VHDL'87)
subprgm	fcopy93.vhd	4.4-2	Procedure to copy a file (VHDL'93)

Directory Name	File Name	Figure #	Description
subprgm	file1.txt, file2.txt, file3.txt, fileout.txt	---	Files used for f_arry87.vhd file
subprgm	fline87.vhd	4.5	Multiple access of a file in VHDL'87
subprgm	image.vhd	4.2-1	Non-leaky dynamically linked object in a subprogram
subprgm	nrmalias.vhd	4.8-3	Normalization using aliases
subprgm	proc87.vhd	4.3.1	Corrected subprogram for VHDL'87
subprgm	proc93.vhd	4.3.2	Corrected subprogram for VHDL'93
subprgm	sidefct.vhd	4.1.1-2	Architecture with side effects
subprgm	timecnvt.vhd	4.7	Conversion package for type time
package	arithstd.vhd	---	Sample application of Std_Logic_Arith package
package	complex.vhd		Application example of complex package
package	complexp.vhd		Complex package
package	image_pb.vhd	5.1	Image package
package	lfsr4.vhd	5.4-4	Design of a four bit LFSR register
package	lfsr_tb.vhd	5.4-6	Testbench for the LFSR function
package	lfsrarth.vhd	---	LFSR package for Std_Logic_Arith Signed and Unsigned
package	lfsrbit.vhd	---	LFSR Bit
package	lfsrnbit.vhd	---	LFSR Numeric_Bit
package	lfsrnstd.vhd	---	LFSR package - Numeric_Std
package	lfsrstd.vhd	5.4-2	LFSR package
package	misr_pb.vhd	5.3	Multiple Input Signature Register (MISR) package
package	misr_tst.vhd	---	Test bench for MISR package
package	numstd.vhd	---	Operations using Numeric_Std (with error)
package	numstdok.vhd	---	Operations using Numeric_Std
package	rand_ea.vhd	---	application of integer random number package
package	rndm.vhd	---	Random integer package
package	rndm_ea.vhd	5.5-2	Application of integer random package
package	rndm_pb.vhd	4.5-3	RND (random) package
package	rndm_int.vhd	5.5-1	Integer random package
package	strgimg.vhd	5.2-2	Use of 'image in VHDL'93
package	strngout.txt	---	simulation results
package	strngout.vhd	5.2-1	Strings to screen
package	uniform.vhd	---	Integer random number procedure
package	vs_util.vhd	---	Conversion routines
package	xyz.in	---	Initialization file
package	file2.vhd	5.2.1	Writing to One file from Multiple Processes
package	deferrpb.vhd	5.6	Referencing Deferred Constant
package	complexp.vhd	5.7-1	Complex Package
models	ej_cfg.vhd	6.3-8	configuration declarations for error injector test
models	ej_cpu.vhd	6.3-5	Simple CPU BFM
models	ej_order.txt	---	Error injector compilation order
models	ej_reg.vhd	6.3-5	Register file BFM
models	ej_tb.vhd	6.3.7	Error injector testbench
models	erini_tb.vhd	---	Error injector testbench
models	errinj.vhd	6.3-4	Error injector model
models	memory.vhd	6.1.1	Memory model, initialized at start-up
models	memry0.vhd	---	Simple memory model, no file preload
models	memrytb.vhd	---	memory test bench

Directory Name	File Name	Figure #	Description
models	ramfix.vhd	6.1.3	Fixed array caching RAM model
models	ramfix_c.vhd	6.1.8-2	Ramfix configuration
models	switch1.vhd	6.4-1	One Switch Model
models	\switch2.vhd	6.4-2	Two Switches in Series Model
models	ramlink.vhd	6.1.5-1	Dynamic page caching
models	ramlnk_c.vhd	6.1.8-2	Ramlink configuration
models	rampag_c.vhd	6.1.8-2	Rampage configuration
models	rampage.vhd	6.1.4	Fixed page caching
models	ram_tb.vhd	6.1.8-1	Memory testbench
models	setholdp.vhd	---	setup and hold package
models	switch1.vhd		
models	switch2.vhd		
models	z0quiet.vhd	6.2-3	Resistor model
models	zohm0_ea.vhd	6.2-2	Resistor model
models	zohm_ea.vhd	6.2-4	Resistor bank model
synths	aggreg.vhd	---	Application of aggregates
synths	areaop1.vhd	7.22-1	Model for optimization
synths	areaop2.vhd	7.22-3	Optimized model for area
synths	areaopt.vhd	---	Area optimization model
synths	array1.vhd	---	Enumerated data type index for synthesis
synths	asoc_bad.vhd	---	Subelement association
synths	asoc_err.vhd	7.17-1	Subelement association
synths	asoc_sy2.vhd	7.17-3	Legal VHDL'93 interface association
synths	asoc_syn.vhd	7.17-2	In-whole subelement association
synths	barrel.vhd	7.11-3	Synthesizable barrel shifter with case statement
synths	barrel2.vhd, barrel3.vhd, barrel4.vhd		
synths	barrelnc.vhd	7.11-1	Non-synthesizable barrel shifter
synths	barlsyn1	7.11-2	synthesizable barrel shifter
synths	bitrevp.vhd	7.25-1	Bit reversal process
synths	bitrevs.vhd	7.25-2	Bit reversal with a function
synths	casecst1.vhd	---	Case statement with constant
synths	compart1.vhd	7.23-1	Model of a comparator
synths	compart2.vhd	7.23-2	Address range identification without inference of a comparator
synths	count.vhd	7.3-2	One-dimensional array declaration and synthesis
synths	countvar.vhd	7.3-3	One-dimensional array declaration with variables
synths	d2ff_asn.vhd	7.3-6	Two sequential Flip-flop with asynchronous reset on one FF
synths	demuxp1.vhd	7.10.2	Demultiplexer Model with a Process
synths	demux1.vhd	---	Demultiplexer model
synths	demuxc.vhd	7.10-1	Demultiplexer model
synths	demuxp2.vhd	7.10-3	Demultiplexer model with a process
synths	dff.vhd	7.6-1	Flip-flop model
synths	dffasync.vhd	7.3-5	D flip-flop with asynchronous reset
synths	dffsync.vhd	7.3-4	D flip-flop with synchronous reset
synths	initvc1.vhd	7.5.1-1	Non-synthesizable VHDL code
synths	initvc2.vhd	7.5.1-1	Synthesizable VHDL code
synths	invert.vhd	---	Inverter model
synths	inverttb.vhd	---	Test bench for inverter
synths	ivtrol.vhd	7.17.1-1	Model of component with unconstrained interfaces
synths	ivtroltb.vhd	7.17.1-2	Subelement association of a component

Directory Name	File Name	Figure #	Description
synths	lead1.vhd	7.12.1-1	Using case statement
synths	lead2.vhd	7.12.1-2	Nested case statement
synths	lead3.vhd	7.12.1-3	Case statement with type "unsigned"
synths	lead4.vhd	7.12.2	Using If instead of case statement
synths	lead5.vhd	7.12.3-1	Using for..loop in a function
synths	lead6.vhd	7.12.3-2	Using for..loop in a process
synths	lead7.vhd	7.12.4	Application of Std_Match function
synths	lead8.vhd	---	Leading one
synths	mc14532b.vhd	7.13.1-2	Parameterized Priority Encoder
synths	mc14532q.vhd	7.13.1-2	Parameterized Priority Encoder
synths	multy.vhd	7.27	Multiplier model
synths	mux.vhd	7.9	Multiplexer model
synths	prech1.vhd		
synths	prechrg.vhd	7.14-4	Precharge Model
synths	prechtb.vhd		
synths	priority.vhd	7.13.2-2	Two-Level Encoding Architecture for Priority Encoder
synths	prty_tb.vhd	7.13.2-3	Priority Encoder Testbench
synths	reg9.vhd	7.7	Synthesis error
synths	regfile.vhd	7.8.1	Register file model
synths	regfilev.vhd	7.8-3	Register File with Variable
synths	regf_bad.vhd	7.8.2-1	Improper Register File with Variable
synths	regok.vhd	7.7.1-2	Corrected shift register model
synths	regok2.vhd	7.7.1-3	Improved shift register model
synths	sharing.vhd	7.21-2	Automatic resource sharing
synths	sharingf.vhd	7.21-3	Resource sharing by design
synths	timer_ea.vhd	7.26-1	Synthesizable timer design – integer method
synths	timr3_ea.vhd	7.26-2	Synthesizable timer design – Unsigned method
synths	togndvcc.vhd	7.24-1	Element association to Ground and Vcc
synths	triasync.vhd	---	Asynchronous tri-state
synths	tr ierr.vhd	7.16-1	Incorrect implementation of tri-state
synths	tr iok.vhd	7.16-2	Corrected model of tri-state with if
synths	tr iok2.vhd	7.16-3	Corrected model of tri-state with case
synths	trisync.vhd	7.16-4	Tri-state enable synchronous control
synths	trisync2.vhd	7.16-6	Method to avoid registering of tri-state control
verific	alu_ea.vhd	8.6-5	ALU model (for test)
verific	alu_tb.vhd	8.6-7	testbench
verific	protocol.vhd	8.6-6	Protocol component
verific	types_p.vhd	8.6-3	Package for definition of a type
verific	uniform.vhd	8.6-4	Integer based random number package
potpouri	cntrcfg.vhd	9.9.9-3	Counter Configuration
potpouri	cntrtb.vhd	9.9.9-2	Counter Testbench
potpouri	cntr_ea.vhd	9.9.9-1	Counter architectures
potpouri	const.vhd	9.1.4	Use of constant for type conversion
potpouri	countdd.vhd	---	Vital 2.2 counter
potpouri	counter.vhd	---	Vital 2.2 counter
potpouri	crash.vhd	---	Model to test host memory capabilities
potpouri	crystal.vhd	9.1.2.2-2	Acceleratable process
potpouri	datain.txt	---	Text files for test
potpouri	dataout.txt	---	Text files for test
potpouri	dataout0.txt	---	Text files for test
potpouri	faq1.txt	---	Internet FAQ1 – general

Directory Name	File Name	Figure #	Description
potpouri	faq2.txt	---	Internet FAQ2 – books
potpouri	faq3.txt	---	Internet FAQ1 - Products and services
potpouri	faq4.txt	---	Internet FAQ1 - VHDL Glossary
potpouri	finta87.vhd	---	Write binary file, VHDL'87
potpouri	finta93.vhd	9.1.5-1	Write binary file, VHDL'93
potpouri	fintb87.vhd	---	Read binary file, VHDL'87
potpouri	fintb93.vhd	9.1.5-2	Read binary file, VHDL'93
potpouri	fline87.vhd	9.3-2	Transfer a line with a signal
potpouri	fline93.vhd	9.3-1	Transfer a line with shared variables VHDL'93
potpouri	global.vhd	---	Use of global signals
potpouri	hierglob.vhd	9.2.1-2	Global signals to access signals internal to component
potpouri	hierport.vhd	9.2.2-1	Use of ports to access signals internal to component
potpouri	rbool.vhd	9.9.11	Resolved Boolean Function and Example
potpouri	redund.vhd	9.1.2.5-1	Example of redundant code
potpouri	redund0.vhd	9.1.2.5-2	Redundant code avoidance
potpouri	redund2.vhd	---	Redundant code avoidance
potpouri	simout.txt	---	Text file for test
potpouri	speed.vhd	9.1.2.2-1	Sensitivity list versus wait statement
potpouri	subprgm.vhd	9.9.12	Subprogram Example
potpouri	types_p.vhd	9.9.3	Example of Type and Constant Declarations
potpouri	wait.vhd	9.1.2.2-2	acceleratable process with wait statement
reuse	agr.vhd	10.2.4	Defining and Using Parameters for Design Parameterization
reuse	global.vhd	10.2.3	Limitations of Globally Static Parameters
reuse	param3.vhd	10.2.5-2	Use of Subtypes and constants
ieee	numbit.vhd	---	Numeric_Bit package DECLARATION
ieee	numstd.vhd	---	Numeric_Std package DECLARATION
ieee	stdlogic.vhd	---	Std_Logic_1164 package
std	standard.vhd	---	Package standard
std	textio.vhd	---	TextIO package
std	vhdl_87.txt	---	Syntax VHDL'87
std	vhdl_93.txt	---	Syntax VHDL'93
synopsys	attribut.vhd	---	ATTRIBUTES package
synopsys	bvarith.vhd	---	BV_ARITHMETIC package
synopsys	slmisc.vhd	---	std_logic_misc package
synopsys	std_sign.vhd	---	STD_LOGIC_SIGNED package
synopsys	std_unsg.vhd	---	STD_LOGIC_UNSIGNED package
synopsys	stdarith.vhd	---	STD_LOGIC_ARITH package
synopsys	stdtxio.vhd	---	STD_LOGIC_TEXTIO package
synopsys	synopsys.vhd	---	SYNOPSYS package
util	random_p.vhd	---	Random package
util	vs_util.vhd	---	Various utilities package
util	vsp.vhd	---	String operations package

# NOTATION CONVENTIONS

The following symbols and syntactic description are used to facilitate the learning of VHDL.

## SYMBOLS

-  Methodology and guideline.
-  Two thumbs up. Good methodology or approach.
-  Two thumbs down. Poor methodology or approach.
-  Disagreement in community on methodology or approach.
-  Legal or OK code
-  Coding Error
-  Synthesizable
-  Non-Synthesizable
- ... Ellipsis points in code: Source code not relevant to discussion.
- [1] Quotations reprinted from IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual (LRM). Quotations printed in "*italic and in this font*".

**Boldface** Boldface in text: Emphasizes important points.

Boldface in syntax and sample code: Emphasizes VHDL reserved words.

*Italic font* Italic in text emphasizes VHDL word versus English text. For example, Objects of type *Signed* or *Unsigned* can be converted to type *Std\_Logic\_Vector* with the VHDL provided type conversion feature.

*Italic* *Italic wide font used in chapter 7 to identify material, extracted by permission, from the "1996 Synopsys Users Group Conference."*

## **SYNTACTIC DESCRIPTION**

**left\_hand\_side ::= right\_hand side**

left\_hand\_side is the syntactic category

right\_hand\_side is a replacement rule

::= (read as "can be replaced by")

**Vertical bar separates alternative items**

*Example: letter\_or\_digit ::= letter | digit*

**Square brackets [] enclose optional items**

*Example: return\_statement ::= return [expression]*

**Braces {} enclose a repeated item (zero or more times).**

*Example: index\_constraint ::= (discrete\_range, {discrete\_rang})*

**Underlined identifies that the notation is applicable for VHDL'93 ONLY**

*Example: ... end configuration [configuration\_simple\_name]*

## Acknowledgments

*VHDL Answers to Frequently Asked Questions* evolved from discussions on *comp.lang.vhdl* newsgroup, European Space Agency (ESA) reports, discussions with Model Technology Inc., Synopsys® presentation, and class discussions.

I especially thank the engineers, consultants, and educators of the *comp.lang.vhdl* public domain newsgroup who have addressed some very interesting questions and furnished challenging answers. These newsgroup contributors supplied a wealth of experiences and wisdom in the use of the language and its practical applications. Their conversations provided the stimulus that became the basis for this book.

I thank the Institute of Electrical and Electronics Engineers, Inc. for granting me permission to extract some material from the IEEE Std 1076-1993 Standard VHDL Language Reference Manual.

I thank Model Technology for the use of their friendly VHDL and Verilog PC and workstation simulation environment, for the loan of their VHDL/Verilog PC compiler/simulator, and for their excellent and competent product support group. Vince Corbin of Model Technology Inc. was particularly helpful in reviewing the book and giving excellent suggestions.

I thank Synopsys, Inc. for the release of their VHDL packages and for granting me permission to incorporate *VHDL Synthesis Techniques and Recommendations*, presented at the 1996 *Synopsys Users Group Conference*. Joseph Pick of Synopsys, Inc., author of that excellent tutorial presentation, reviewed this book and furnished many technical and editorial suggestions.

Sandi Habinc<sup>2</sup> from the European Space Agency published *VHDL Models for Board-Level Simulation* that was used as a seed in the discussions on verification and simulation speed. I thank Sandi for granting me permission to extract some valuable information from this document, and for supplying important comments.

I thank Dino Caporossi, *VFormal* Product Manager at COMPASS Design Automation, for contributing the article on formal verification.

I thank the following reviewers of this book who have provided valuable comments and useful suggestions in the first edition of this book: Paul Menchini of *Menchini & Associates*, Richard Hall of *Cadence Design Systems, Inc.*<sup>3</sup>, Jake Karrfalt of *Alternative System Concepts, Inc.*,<sup>4</sup> Greg Peterson, LT USAF under the direction of John Hines from USAF, and Johan Sandstrom and Stephen Schoessow, modeling and synthesis consultants.

I thank Carl Harris from Kluwer Academic Publishers for supporting me in this book endeavor. I acknowledge my daughter Lori Hillary, and my son Michael Lloyd for inspiring me to teach.

## About the Author

**Ben Cohen** has an MSEE from USC and is a Scientist engineer at Hughes Aircraft Company. He has technical experience in digital and analog hardware design, computer architecture, ASIC design, and synthesis. He used hardware description languages for the modeling of systems at various modeling levels including, statistical, instruction set architecture, and synthesis. He applied VHDL, since 1990, to model various bus functional models of computer interfaces. He was also involved on several hardware, software, and firmware tasks for several microprocessor applications. He taught classes at USC and at Hughes on the application of microprocessors, Pascal, and VHDL. In 1995, he authored the book *VHDL Coding Styles and Methodologies*, ISBN 0-7923-9598-0 Kluwer Academic Publishers. In 1996, he authored the book *VHDL Answers to Frequently Asked Questions* (1<sup>st</sup> edition), ISBN 0-7923-9791-6 Kluwer Academic Publishers.

He recently studied and applied SES Workbench<sup>5</sup>, a statistical and transaction based system level simulator, for the analysis of systems when exposed to application workloads.

email:        [VhdlCohen@aol.com](mailto:VhdlCohen@aol.com)  
ftp:        [users.aol.com/vhdlcohen/vhdl](http://users.aol.com/vhdlcohen/vhdl)  
Web:        <http://members.aol.com/vhdlcohen/vhdl>

---

**VHDL ANSWERS TO FREQUENTLY  
ASKED QUESTIONS**  
**Second Edition**

## **1. LANGUAGE ELEMENTS**

---

---

This section addresses questions related to the basic language elements of VHDL. It includes discussions on the salient points of concurrent statements, configurations, ports, arithmetic issues, and package Std\_Logic\_1164. Synthesis is also addressed because of its importance in the design of ASICs and FPGAs.

This section also defines a method to allow multiple processes to access exclusive use of shared variables.

## VHDL Elements and Inter-relationships

**Q**

What are the VHDL language elements and how are they inter-related?

**A**

The elements include:

- Types
- Attributes
- Entities
- Operators
- Concurrent Statements
- Architectures
- Subprograms
- Packages
- Configurations
- Objects
- Libraries

The elements used in the architectural planning and gate level implementation of an electronic component can be compared to the elements used in the construction of a house. Table 1.1 provides a one-to-one comparison and a simplistic approach into the understanding of language elements. These fundamental elements are used in the construction process of any design, whether it be an architecture of a dwelling or a model of an electronic component. Figure 1.1 provides a graphical representation of the inter-relationships between VHDL elements.

**Table 1.1 Basic Elements of a Design**

HOUSE	VHDL
<ul style="list-style-type: none"> <li>• MATERIAL TYPES           <ul style="list-style-type: none"> <li>- Nails, Beams, Plywood, Pipes, ...</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• TYPES           <ul style="list-style-type: none"> <li>Integer, Real, Record, Std_Logic_Vector, ...</li> </ul> </li> </ul>
<ul style="list-style-type: none"> <li>• OBJECTS (an instance of a type)           <ul style="list-style-type: none"> <li>- Two-by-fours by 8 ft, 3/4" copper pipes, ..</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• OBJECTS (an instance of a type)           <ul style="list-style-type: none"> <li>- Variables, signals</li> </ul> </li> </ul>
<ul style="list-style-type: none"> <li>• ATTRIBUTES           <ul style="list-style-type: none"> <li>- Nail size, Beam material, ...</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• ATTRIBUTES           <ul style="list-style-type: none"> <li>- My_Signal'length, My_Var'range, ...</li> </ul> </li> </ul>
<ul style="list-style-type: none"> <li>• TOOLS (to operate on objects)           <ul style="list-style-type: none"> <li>- Hammer (Addition, connection)</li> <li>- Saw (Division, subtraction)</li> <li>- Tape (equality, inequality) ...</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• OPERATORS (to operate on objects)           <ul style="list-style-type: none"> <li>- "+", "&amp;", ...</li> <li>- "-", "/", "*"</li> <li>- "="; "/=", "&gt;="; ..</li> </ul> </li> </ul>
<ul style="list-style-type: none"> <li>• BUILDING CODES / PROCEDURES</li> </ul>	<ul style="list-style-type: none"> <li>• SUBPROGRAMS</li> </ul>
<ul style="list-style-type: none"> <li>• CUSTOM MADE THINGS           <ul style="list-style-type: none"> <li>- Tools, Doors, Windows, ...</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• PACKAGES           <ul style="list-style-type: none"> <li>- Operators, Subprograms, Objects</li> </ul> </li> </ul>
<ul style="list-style-type: none"> <li>• CONSTRUCTION WORKERS</li> </ul>	<ul style="list-style-type: none"> <li>• CONCURRENT STATEMENTS</li> </ul>
<ul style="list-style-type: none"> <li>• EXTERIOR DRAWINGS / PLANS</li> </ul>	<ul style="list-style-type: none"> <li>• ENTITIES</li> </ul>
<ul style="list-style-type: none"> <li>• FLOOR PLAN /DETAILED ARCHITECTURE</li> </ul>	<ul style="list-style-type: none"> <li>• ARCHITECTURE</li> </ul>
<ul style="list-style-type: none"> <li>• PLAN DEFINITIONS FOR SAME EXTERIOR</li> </ul>	<ul style="list-style-type: none"> <li>• CONFIGURATIONS</li> </ul>
<ul style="list-style-type: none"> <li>• PREFABRICATED MATERIALS           <ul style="list-style-type: none"> <li>- Windows, Tubs, Showers, Faucet</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• LIBRARIES and COMPONENTS           <ul style="list-style-type: none"> <li>- Compiled entities, architectures, configurations, packages</li> </ul> </li> </ul>
<ul style="list-style-type: none"> <li>• FOUND IN           <ul style="list-style-type: none"> <li>Home Improvement Stores</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• FOUND IN           <ul style="list-style-type: none"> <li>- Packages and libraries</li> </ul> </li> </ul>

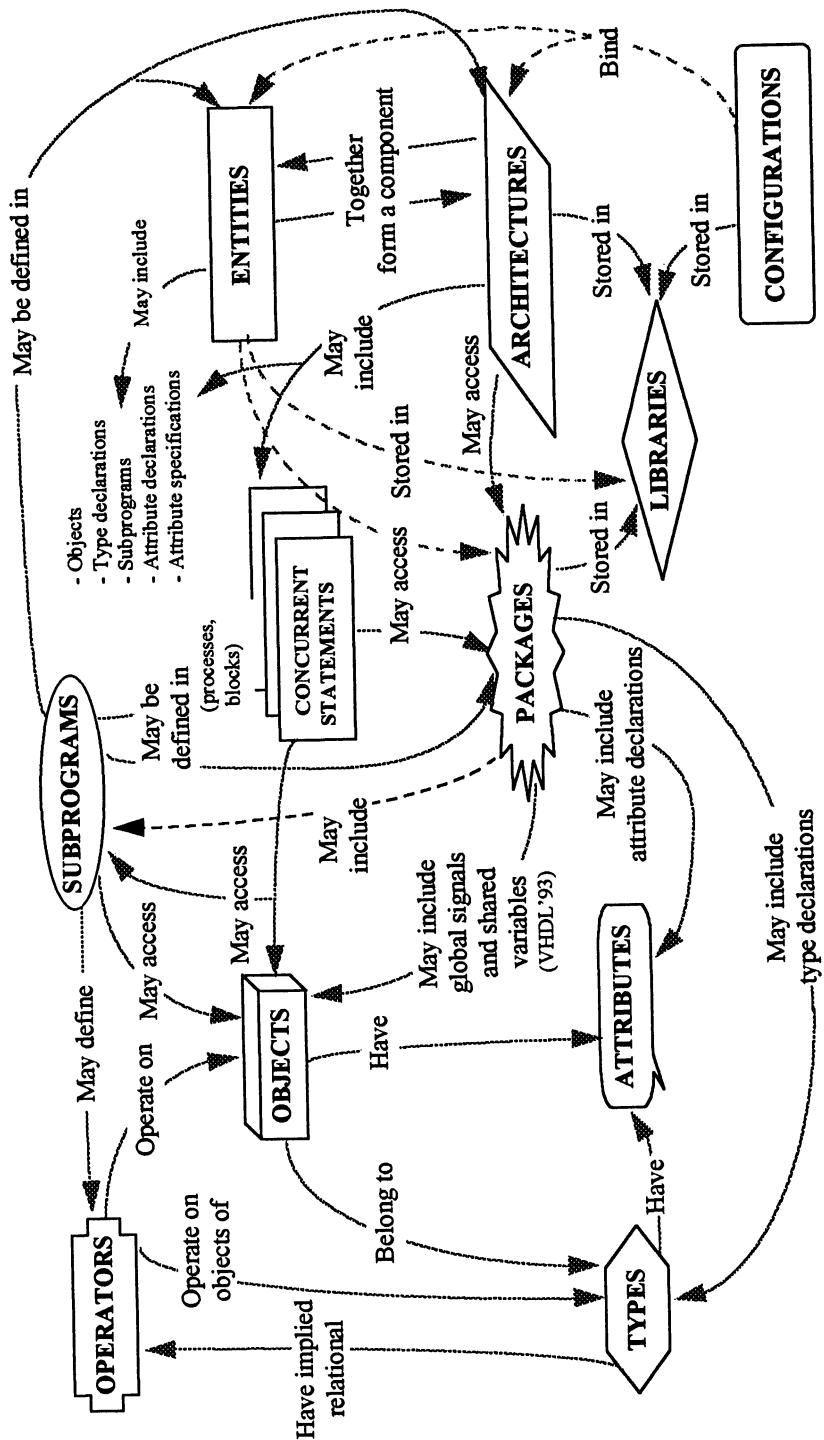


Figure 1.1 Inter-Relationships between VHDL Elements

## 1.1 WHY VHDL FOR DIGITAL DESIGNS

**Q** | Why is VHDL applicable to digital designs? How does it compare to graphical design capture?

**A** | Historically, logic design progressed from manual design capture on vellum, to computer aided design using workstations. Both of these approaches were very structural in nature where the primitive components typically represented commercially available or custom devices. As technology progressed and as designs became significantly more complex, synthesis tools started to play a very significant role in the design process. Portions of the designs (typically the combinatorial logic) were captured in a hardware description language (tables, or vendor specific format). These designs were synthesized into blocks that were integrated into the design using the graphical design tools. The graphical netlist represented the structural elements (such as flip-flops, buffers) and the synthesized combinatorial logic blocks. That netlist was then synthesized into the desired components using synthesis tools.

As hardware description languages (such as VHDL and Verilog) and synthesis tools matured, it became apparent that whole designs could be captured without graphical design tools. Below is a summary of the advantages in using VHDL.

### **VHDL ADVANTAGES**

- **Clear definition of design requirements**

VHDL allows for multi-level (e.g., behavioral, RTL, structural) and hierarchical definitions of a design.

- **Efficiency in design cycle time**

The design can be defined at the functional level, with less concern to minimization, and low level implementation (e.g., counters, adders). The design can rely on synthesis tools to provide and help optimize the low level implementation

- **Flexibility in adapting to design changes**

VHDL allows for fast coding changes using a text editor. A one line change may impact 50+ connections. VHDL allows for design parameterization (see chapter 10).

- **Reuse of designs and packages**

VHDL enables easy design reuse and adaptability using custom or commercial libraries.

- **Technology independent**

VHDL is technology independent. The synthesis tools are responsible for mapping the design requirements into the required technology using the specified libraries.

- **Easy analysis of various architectures/implementations**

The transition between design definition and simulation is fast. VHDL allows the use of configurations to select various combinations of architectures for the components.

- **Design verification and auto-regression tests**

VHDL enables the quick and thorough definition of testbenches (environment around the unit under test). VHDL allows the capability to drive the UUT with pseudo-random, or externally generated data. VHDL enables the definition and integration of verification monitors for automatic and thorough error detection and reporting (see chapter 8).

- **VHDL is recommended for government contracts**

- **VHDL commercial models are available for purchase**

- **VHDL is a documentation language**

- **VHDL is a simulation language**

### **Schematic Capture Versus VHDL**

- **Schematic capture is slow.**

VHDL is textual, and can be edited, searched, and modified very quickly using a language sensitive editor (such as *emacs*) and a coding style that lends itself to reusability (see chapter 10). Schematic capture is slow, and changes are tedious.

- **Design is technology dependent.**

VHDL is technology independent, whereas schematic capture generally requires a library of specific components.

- **Readability is weak.**

Schematic designs span over multi-sheets. The functionality of the design may not necessarily be obvious from a structural level implementation.

- **Adaptability to design changes is weak.**

In schematic capture, a change can be very significant, and maybe reflected in many design sheets. For example, a change in a control field may require several changes. In VHDL, this change may represent a single line change in the subtype definition (see chapter 10).

- **Design optimization must start from the start and is manual**

Because of the difficulty in making design changes, users tend to optimize the design while it is being shaped. Optimization is best performed throughout the design process, with the help of such tools as synthesis, timing and signal integrity analyzers.

- **Design reuse is weakened**

Designs captured with schematic capture tools are not easily portable. In many cases, vendors have upgraded their tools, but have not maintained downward compatibility. Thus, it is a very tedious process to translate a design to the newer versions. However, VHDL is textual based, and thus only requires an ASCII text editor.

- **Verification process is weak and non-portable**

Verification process through simulation is usually performed with tools supplied by the vendor supporting the design capture. VHDL simulations are tool independent, and any VHDL compliant simulator can be used.

## 1.2 SALIENT POINTS OF CONCURRENT STATEMENTS

**Q** | What are the salient points of concurrent statements?

**A** | [1] A concurrent statement is a statement that executes asynchronously, with no defined relative order. Concurrent statements are used for dataflow and structural descriptions.

All concurrent statements execute concurrently with respect to each other. During any activation time (e.g., event on a sensitivity list, end of timeout period), each statement executes sequentially until it reaches a *wait* condition (implicit or explicit). The simulator then jumps to the next concurrent statement that needs to be executed. When ALL the concurrent statements for a particular time are completed, time advances to the next activity. The concurrent statements include the following:

- Process
- Concurrent signal assignment
- Component instantiation
- Concurrent procedure
- Generate
- Concurrent assertion
- Block

### 1.2.1 Process

- Processes are very important because they allow sequential statements, like loops and conditional statements (e.g., *if*, *case*). They allow the use of variables that are updated immediately, and occupy much less simulator memory than signals (two orders of magnitude less, depending on the simulator).
- Processes with sensitivity lists have an implicit wait on signals\_in\_sensitivity\_List statement at the end of the process. A process with a sensitivity list cannot have explicit *wait* statements.
- A process must have a *wait* statement (explicit or implicit) in all the paths (e.g., *if* or *case* statements) within the process. Otherwise, the process will loop forever, and time will never advance. This condition may not necessarily be detected by the compiler. This is demonstrated in figure 1.3.1.

```

architecture BadProcess_a of BadProcess is
  signal S : Bit;
begin
  BadProcess_Lbl: process
  begin
    if S = '0' then
      S <= '1';
      wait for 10 ns;
    end if;
    -- Note: if S = '1' then process loops forever,
    --         and time does advance.
  end process BadProcess_Lbl;
end BadProcess_a;

architecture OkProcess_a of BadProcess is
  signal S : Bit;
begin
  OkProcess_Lbl: process
  begin
    if S = '0' then
      S <= '1';
      wait for 10 ns;
    else
      wait for 10 ns; -- ☺ All conditions have a WAIT
    end if;
  end process OkProcess_Lbl;
end OkProcess_a;

```

Figure 1.3.1 Wait statements in Process (element\process.vhd)

## 1.2.2 Concurrent Signal Assignment Statements

- Concurrent signal assignments are very useful to describe combinational logic. There are two flavors:

- Conditional signal assignment (similar to *if* in a process). For example:

```

S <= '1'      when A = '0' else
              In2      when B = '1' else
              not In2;

```

- Selected signal assignment (similar to *case* in a process). For example, in figure 1.3.2-1

```

architecture T_a of T is
  signal A : Bit;
  signal B : Bit;
  signal S : Bit;
begin
  with Bit_Vector'(A, B) select
    S <= '1'      when "00",
    '0'          when "01",
    '0'          when others;
end T_a;

```

(A, B) is an aggregate. It is preceded with a type qualifier to clearly identify the type. The length of the aggregate is determined by the aggregate components in a manner similar to the statement:  
**constant C : Bit\_Vector := "1011";** – length from aggregate

Figure 1.3.2-1 Selected Signal Assignment (element\ssasgnt.vhd)

- Concurrent signal assignments are sensitive to ALL signals on the right hand side of the signal assignment. In above examples, the concurrent signal assignments are sensitive to signals *A*, *B*.
- Concurrent signal assignments are useful to generate waveforms (see figure 1.3.2-2). However, when the waveforms are extensive, a process with a waveform generator algorithm, or a design of a bus functional model (section 8.5) is a preferred approach.

```

architecture Wave_a of Wave is
  signal A : Std_Logic_Vector(3 downto 0) := (others => 'Z');
  signal C : Std_Logic_Vector(3 downto 0) := (others => 'Z');
  signal S1 : Std_Logic := 'Z';
begin
  Aggregate :>
  (S1, A(0), C(2), C(1)) <=
    Std_Logic_Vector'("1010"),
    Std_Logic_Vector'("0001") after 10 ns,
    Std_Logic_Vector'("1100") after 20 ns,
    Std_Logic_Vector'("0101") after 30 ns,
    Std_Logic_Vector'("0010") after 40 ns,
    Std_Logic_Vector'("1001") after 50 ns;
end Wave_a;

```

Type qualifier to identify that string literal is of type Std\_Logic\_Vector.

Simulation Results				
ns	delta	A	C	S1
0	+0	ZZZZ	ZZZZ	Z
0	+1	ZZZ0	Z10Z	1
10	+0	ZZZ0	Z01Z	0
20	+0	ZZZ1	Z00Z	1
30	+0	ZZZ1	Z01Z	0
40	+0	ZZZ0	Z10Z	0
50	+0	ZZZ0	Z01Z	1

Figure 1.3.2-2 Waveform Generation (element\wave.vhd)

### 1.2.3 Component Instantiation

- A component represents an entity and architecture pair.
- A component represents a level of hierarchy.
- A component can be instantiated one or more times in an architecture (just like real hardware).
- If the name of the component is the same as the name of the entity, then the default binding is the latest compiled architecture for that entity.
- Configurations can be used to explicitly bind a component to an entity and an architecture.
- At component instantiation, connection of ports to signals must be made (like connection from component pins to traces of a board). There are rules that define when a port can be associated with an *open* (see section 1.5.3).
- At component instantiation, type conversion between the formal and actual parameters of ports can be made (see section 1.5.4).

### 1.2.4 Concurrent Procedure

- A concurrent procedure represents a procedure that is instantiated as a concurrent statement. It is equivalent to a process with a procedure call followed by an explicit wait on sensitivity list. The sensitive signals are those extracted from all the actual signals whose modes in the formal parameter list are *in* or *inout*.
- A concurrent procedure with side effects will lose the implied sensitivity lists on those side effect signals. This is because a concurrent procedure is not sensitive to signals

with side effects. Figure 1.3.4 demonstrates this concept.

```

architecture CProc_a of Cproc is
    signal S : Boolean;

    procedure P(A : Boolean) is
        begin
            assert S
                report "S is False -- Procedure P"
                severity Note;
        end P;

    procedure P2(signal A_s : Boolean) is
        begin
            assert A_s
                report "S is False -- Procedure P2"
                severity Note;
        end P2;

begin
    P(False);
    P2(S);
    S <= True,
        False after 10 ns,
        True after 20 ns,
        False after 30 ns;
end Cproc_a;

```

Procedure has SIDE EFFECTS on signal "S". Procedure is NOT sensitive to signal "S". ⚡⚡

Procedure has NO side effects. Procedure is sensitive to ACTUAL signal PASSED in the procedure call. ⚡⚡

**SIMULATION RESULTS**

```
# ** Note: S is False -- Procedure P2
# Time: 0 ns Iteration: 0 Instance:/
# ** Note: S is False -- Procedure P
# Time: 0 ns Iteration: 0 Instance:/
# ** Note: S is False -- Procedure P2
# Time: 10 ns Iteration: 0 Instance:/
# ** Note: S is False -- Procedure P2
# Time: 30 ns Iteration: 0 Instance:/
```

**Figure 1.3.4 Concurrent Procedure with Side Effect (element\cproc.vhd)**

- Concurrent procedures can be instantiated multiple times.
- Variable parameters are not allowed in a concurrent procedure call.

### 1.2.5 Generate

- The generate statement allows for the conditional or iterative instantiation of concurrent statements at elaboration time.
- The generate statement is very useful to generate or build structures that are dependent upon a generic value, or that are semi-regular (like a parity tree). See section 1.6.4 and 6.2 and 6.3 for examples.
- The generate statement is synthesizable.

### 1.2.6 Concurrent Assertion

- The concurrent assertion statement is used to alert a user of a condition, such a violation. The message is displayed on the output display or screen.
- The sequential assertion statement is of the same form as the concurrent assertion statement used in a process.

### 1.2.7 Block statement

The syntax for a block is shown below:

```
block_statement ::=  
    block_label : block [(guard_expression)]  
    [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 ; ] ]
```

Guard expression are not supported by synthesis



Block header is not necessarily supported by synthesis



- The block statement is used in VHDL Initiative Toward ASIC Libraries (*VITAL*). See section 9.7 for more information on *VITAL*.
- Blocks are synthesizable provided signal kind *register* and *bus* and block guards are not used. Use type *Std\_Logic* or *Std\_Logic\_Vector* instead of block guards.
- In synthesis, blocks can be used as a separate level of hierarchy to force the synthesizer to compile and optimize sections of the design separately. A complex architecture may present problems in synthesis, layout, and code readability. Partitioning an architecture into blocks provides the following advantages:
  1. Faster synthesis compilation time. Synopsys® provides a *group -hdl\_block* directive that groups design partitions and creates new levels of hierarchies. This approach can significantly reduce the compilation time.
  2. Information hiding. Within a block, local type and signal declarations can be defined. The signals within the block are local to the block, and are not visible by the architecture.
  3. Declaration of partition interfaces. The block header enables the definition of local generics, ports, and port maps. Block headers are not necessarily supported by all synthesizer vendors.
  4. Visibility into architecture. Since a block is a concurrent statement of an architecture, a block has visibility into the ports and signals of the architecture and into the declarations of the component within the architecture. A block also has visibility into packages, constants, and types declared in the entity and architecture. Figure 1.3.7-1 provides an example of a block for a simple design. Figure 1.3.7-2 represents the same example, but includes a block header. Both examples synthesize in *Synplify-Lite* with *QuickLogic*<sup>[8]</sup>.

```
entity BlkExmpl is  
    port(A      : in     Bit_Vector(2 downto 0);  
          B      : out    Bit;  
          C      : out    Bit);  
end BlkExmpl;
```

```

architecture BlkExmpl_a of BlkExmpl is
    signal T_s : Bit;
begin
    Logic_Lbl: block
        signal Local_s : Bit;
        begin
            -- Concurrent statements
            Local_s <= A(2) and A(1);
            T_s <= not Local_s;
            B <= not A(0);
        end block Logic_Lbl;

        -- Concurrent statement
        C <= T_s or A(0);
end BlkExmpl_a;

```

Architectural signal visible by ALL concurrent statements of architecture

Local block signal visible by All concurrent statements of the BLOCK

Figure 1.3.7-1 Block Example in Synthesis (element\block.vhd)

```

architecture BlkExmpl_a of BlkExmpl is
    signal T_s : Bit;
begin
    Logic_Lbl: block
        port(Data : in Bit_Vector(2 downto 0);
              OutB : out Bit;
              Temp : out Bit);

        port map
            (Data => A,
             OutB => B,
             Temp => T_s);
        signal Local_s : Bit;
        begin
            -- Concurrent statements
            Local_s <= A(2) and A(1);
            Temp <= not Local_s;
            OutB <= not Data(0);
        end block Logic_Lbl;

        -- Concurrent statement
        C <= T_s or A(0);
end BlkExmpl_a;

```

Port header with port declaration and port map. The port header may not be synthesizable.



Figure 1.3.7-2 Block Example with Port Header (element\blockh.vhd)

### 1.3 GUARDED SIGNAL ASSIGNMENTS

**Q**

What are guarded signal assignments? When should they be used?

**A**

Per LRM 4.3.1.2 and 9.1, [1] if a signal kind (e.g., register, bus) appears in a signal declaration, then the signals so declared are guarded signals of the kind indicated. Null

can be assigned to guarded signals. Guarded signals can be used in applications where multiple concurrent statements need to either contribute a value onto a signal, or be excluded from such contributions. When a guarded signal is assigned a null, that signal assignment is not considered in the resolution function. This contrasts with non-guarded

signals where, in *Std\_Logic*, the assignment of 'Z' mimics the open contribution or tri-State. However, it does contribute to the resolution function. For example, a CPU with an arithmetic unit, a logic unit, and a floating point unit needs to send results to a local bus when its computations are valid. With guarded signals, each computational unit (modeled as a concurrent statement), can assign a *null* onto the local bus until it is eligible to assert the computed result. Thus, each computational unit is excluded from contributing a value until a non-*null* is assigned. With *Std\_Logic*, each computational unit always contributes a value to the resolution function.

 It is generally recommended NOT to use guarded signals because:

1. Guarded signals are not synthesizable
2. *Std\_Logic* type eliminated the need for guarded statements.
3. Guarded signals are harder to read because the guards and the drivers are not collocated.
4. Users lack familiarity with guarded signals.

However, if guards must be used, then the following summarizes rules regarding guarded statements. [1]

- For a guarded signal that is a composite type, each subelement is likewise a guarded signal.
- A guarded signal is assigned values under the control of Boolean-valued guard expressions (or guards).
- Guards can be implicitly or explicitly defined. 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 must be type *BOOLEAN*. The implicit signal *GUARD* must not have a source. Alternatively, a signal called *GUARD* of type Boolean can be explicitly be declared. That explicit signal can have a source.
- When a given guard becomes False, the drivers of the corresponding guarded signals are implicitly assigned a *null* transaction to cause those drivers to turn off.
- A disconnect specification is used to specify the time required for those drivers to turn off.

The value of signal *GUARD* is always defined within the scope of a given block, and 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.

- In a concurrent signal assignment, 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 its inputs.
- A **register** is a kind of guarded signal that retains its last driven value when all of its drivers are turned off.

Figure 1.4-1 represents an application of guarded statements and its testbench. Figure 1.4-2 represents the simulation results.

```

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.Std_Lock_Unsigned.all;

entity TGuard is
  generic(Width_g : Integer := 32);
  port (In1    : in  Std_Lock_Vector(Width_g - 1 downto 0);
        In2    : in  Std_Lock_Vector(Width_g - 1 downto 0);
        Opr    : in  Std_Lock_Vector(1 downto 0);
        Result : out Std_Lock_Vector(Width_g - 1 downto 0));
end TGuard;

architecture TGuard_a of TGuard is
  constant Plus_c    : Std_Lock_Vector(1 downto 0) := "00";
  constant And_c     : Std_Lock_Vector(1 downto 0) := "01";
  constant MultHi_c  : Std_Lock_Vector(1 downto 0) := "10";
  constant MultLo_c  : Std_Lock_Vector(1 downto 0) := "11";
  signal Result_s   : Std_Lock_Vector(Width_g - 1 downto 0) bus; ← Guarded signal

begin -- TGuard_a
  ALU_Blk: block(Opr = Plus_c) ← Guard signal assigned TRUE if Opr = Plus_c
  begin -- block ALU_Blk
    Result_s <= guarded In1 + In2;
  end block ALU_Blk;

  Logical_Blk: block ← Explicit Guard
  begin -- block Logical_Blk
    Guard <= Opr = And_c;
    Result_s <= guarded In1 and In2;
  end block Logical_Blk;

  -- Process: Multiply_Lbl
  Multiply_Lbl : process(Opr)
    variable Product_r : Std_Lock_Vector((2 * Width_g) - 1 downto 0);
  begin -- process Multiply_Lbl
    Product_r := In1 * In2;
    if Opr = MultHi_c then
      Result_s <= Product_r((2 * Width_g) - 1 downto Width_g);

    elsif Opr = MultLo_c then
      Result_s <= Product_r(Width_g - 1 downto 0);

    else
      Result_s <= null;
    end if;
  end process Multiply_Lbl;

  Result <= Result_s;
end TGuard_a;

library IEEE;
use IEEE.Std_Lock_1164.all;

entity TGuardTB is
  generic(Width_g : Integer := 32);
end TGuardTB;

architecture TGuardTB_a of TGuardTB is
  constant Plus_c    : Std_Lock_Vector(1 downto 0) := "00";
  constant And_c     : Std_Lock_Vector(1 downto 0) := "01";
  constant MultHi_c  : Std_Lock_Vector(1 downto 0) := "10";
  constant MultLo_c  : Std_Lock_Vector(1 downto 0) := "11"; ← Constants to enhance readability

```

```

component TGuard
  generic(Width_g : Integer := 32);
  port (In1    : in  Std_Logic_Vector(Width_g - 1 downto 0);
        In2    : in  Std_Logic_Vector(Width_g - 1 downto 0);
        Opr    : in  Std_Logic_Vector(1 downto 0);
        Result : out Std_Logic_Vector(Width_g - 1 downto 0));
end component;

signal In1    : Std_Logic_Vector(Width_g - 1 downto 0);
signal In2    : Std_Logic_Vector(Width_g - 1 downto 0);
signal Opr    : Std_Logic_Vector(1 downto 0);
signal Result : Std_Logic_Vector(Width_g - 1 downto 0);

begin
  In1 <= "1010101000010101010001111101001" after 100 ns,
         "1111111111111100000000000000000000" after 150 ns,
         "00000000111111000000000000111000" after 200 ns,
         "11111110000000100011100100001111" after 250 ns;

  In2 <= "10000000000000000000000000000111101001" after 100 ns,
         "100000000000000000000000000001110000000" after 150 ns,
         "0011111111000000000000000000111000" after 200 ns,
         "11000000000000111111111111001111" after 250 ns;

process
begin
  Opr <= "00" after 75 ns,
         "01" after 85 ns,
         "10" after 100 ns,
         "11" after 120 ns;
  wait for 150 ns;
end process;

U1: TGuard
  generic map(Width_g => 32)
  port map(In1    => In1,
            In2    => In2,
            Opr    => Opr,
            Result => Result);
end TGuardTB_a;

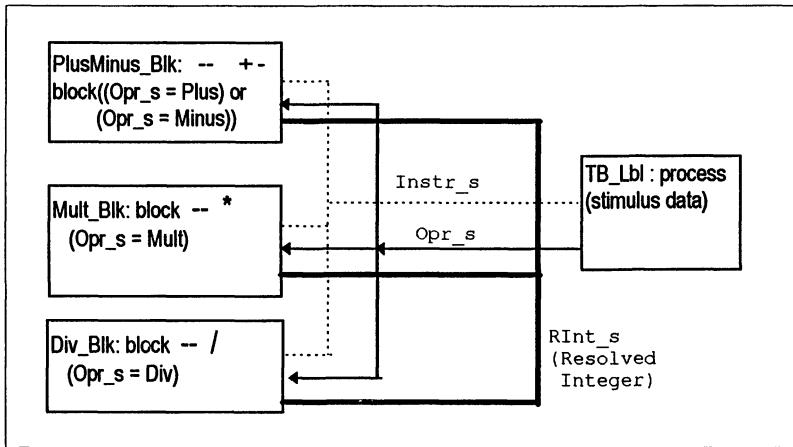
```

**Figure 1.4-1 Application of Guarded Statements and Testbench  
(element\tguard.vhd)**

ns	delta	in1	In2	opr	result	Instruction
0	+2	XXXXXXXX	XXXXXXXX	X	ZZZZZZZZ	
75	+2	XXXXXXXX	XXXXXXXX	0	XXXXXXXX	Plus
85	+3	XXXXXXXX	XXXXXXXX	1	XXXXXXXX	And
100	+3	AA1547E9	800003E9	2	550AA68D	MultHi
120	+2	AA1547E9	800003E9	3	8D362E11	MultLo
150	+0	FFFF0000	80000380	3	8D362E11	
200	+0	00FE0078	3FC00078	3	8D362E11	
225	+2	00FE0078	3FC00078	0	40BE00F0	Plus
235	+3	00FE0078	3FC00078	1	00C00078	And
250	+3	FE02390F	C007FFCF	2	BE899AAC	MultHi
270	+2	FE02390F	C007FFCF	3	6A0B1421	MultLo

**Figure 1.4-2 Simulation Results for Model shown in Figure 1.4-1**

Figures 1.4-3 and 1.4-4 represent another example for the potential use of guarded statements where three functional blocks perform separate operations in data. If an instruction is for a specific block, then that block executes the operation and supplies the result onto a resolved signal of type *integer*. The resolution function for this resolved *integer* is relatively simple since only one driver will drive a value onto the guarded signal (*Rint\_s*) while the other drivers will drive a *null*. That resolution function detects the condition of multiple drivers and reports an error. If there are no drivers, then it returns *Integer'low*. If there is one driver, it then returns the value of that driver.



**Figure 1.4-3 Functional Diagram for T4Guard Model**

```

entity T4Guard is
end T4Guard;

architecture T4Guard_a of T4Guard is
  type Opr_Typ is (Plus, Minus, Mult, Div, None);

  type Instr_typ is record
    Int1 : Integer;
    Int2 : Integer;
  end record;

  type AInteger_Typ is array(Natural range <>) of Integer;

  function Resolve(D : AInteger_Typ) return Integer is
  begin
    assert (D'length = 1 or D'length = 0)
      report "More than one driver of Integer type"
      severity Warning;

    if D'length = 0 then
      return Integer'low;
    else
      return D(D'low);
    end if;
  end Resolve;

  subtype RInteger_Typ is Resolve Integer;

  signal Instr_s : Instr_typ :=
    (Int1 => 5,
     Int2 => 6);
  
```

```

signal RInt_s : RInteger_Typ bus;
signal Opr_s      : Opr_Typ;

begin -- T4Guard_a
  PlusMinus_Blk: block((Opr_s = Plus) or (Opr_s = Minus))
    begin -- block Int_Blk
      Int_Lbl : process(Opr_s)
        begin -- process Flt_Lbl
          if Guard then
            if Opr_s = Plus then
              RInt_s <= Instr_s.Int1 + Instr_s.Int2;
            elsif Opr_s = Minus then
              RInt_s <= Instr_s.Int1 - Instr_s.Int2;
            end if;
          else
            RInt_s <= null;
          end if;
        end process Int_Lbl;
      end block PlusMinus_Blk;
    Mult_Blk: block(Opr_s = Mult)
      begin -- block Int_Blk
        RInt_s <= guarded (Instr_s.Int1 * Instr_s.Int2);
      end block Mult_Blk;
    Div_Blk: block(Opr_s = Div)
      begin -- block Int_Blk
        RInt_s <= guarded (Instr_s.Int1 / Instr_s.Int2);
      end block Div_Blk;
    -----
    -- Process: TB_Lbl
    -- Purpose: Provide stimulus
    -----
    TB_Lbl : process
      begin -- process TB_Lbl
        TestOprLbl: for I in Opr_Typ loop
          Opr_s <= I;
          Instr_s.Int1 <= Instr_s.Int1 + 1;
          Instr_s.Int2 <= Instr_s.Int2 + 1;
          wait for 100 ns;
        end loop TestOprLbl;
      end process TB_Lbl;
    end T4Guard_a ;

```

In a process, one must assign a null if Guard is false

In a concurrent signal assignment , the null is automatically assigned if the Guard is false

Figure 1.4-4 Application of Guards (element\T4Guard.vhd)

```

-- Simulation results
--      ns   delta           instr_s      rint_s opr_s
--      0     +1             (6, 7)       11 plus
--     100    +2             (7, 8)       -1 minus
--     200    +2             (8, 9)       72 mult
--     300    +2             (9, 10)      0 div
--     400    +2             (10, 11)     -2147483648 none
--     500    +2             (11, 12)      23 plus
--     600    +2             (12, 13)      -1 minus
--     700    +2             (13, 14)      182 mult
--     800    +2             (14, 15)      0 div
--     900    +2             (15, 16)     -2147483648 none

```

Figure 1.4-5 Simulation Results for T4Guard Model shown

## 1.4 SIGNALS AND PORTS

### 1.4.1 Disconnecting a Signal

**Q** What is the meaning of disconnecting a signal? How can this signal disconnect be achieved using guard statements?

**A** A signal can be disconnected in a guarded block by assigning a *null* transaction:  
 $S \leq null;$

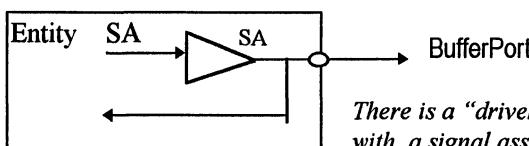
The process executing the assignment of the *null* is no longer driving the signal. Thus, there is one less contributing value that is fed into the resolution function for S. (The exact resolution semantics depends on whether S is declared as a BUS or REGISTER signal kind).

The use of guarded signals and null transactions is discouraged. It is generally recommended to use Std\_Logic\_1164 package that defines the logic state 'Z' (for tri-state), and the resolved types *Std\_Logic* and *Std\_Logic\_Vector*. The use of these resolved types accomplishes the same functionality in a way that is both standardized, synthesizable, and more acceptable to designers. Synthesis tools do not support the *guard* statement, but do support the *Std\_Logic\_1164* package. See section 1.4 for a discussion of guarded signals.

### 1.4.2 Difference between *inout* and *buffer* ports

**Q** What is the difference between *inout* and *buffer*? How are they used?

**A** A buffer is an *OUT port* with read capability. A buffer port may have at most ONE driver within the architecture (i.e., a buffer port can be the only driver on a net).



*There is a "driver" associated with a signal assignment (SA).*

Buffer ports should NOT be used.

*Rationale:* The port direction of a component should reflect the actual modes of the ports, using *in*, *out* and *inout* only. The interface design should be completely separate from the implementation. However, intermediate signals that are declared within the architecture allow both 'read' and 'write' within the same architecture. The values of these internal signals can then be assigned to the *out* ports using concurrent signal assignments (see figure 1.5.2-4). Buffer ports have no correspondence in actual hardware and they impose restrictions on what can be connected to them.

In synthesis, ports of direction *out*, *inout* and *buffer* have NO correlation with the implemented hardware drivers (e.g., push-pull, open collector, or tri-state driver). The hardware driver decision is determined by the values of the signals supplied to the ports

and by the directed technology. If a 'Z' is assigned onto a port, the synthesizer will implement a tri-state or open collector hardware driver. If only forcing values (e.g., '1' and '0') are assigned, with no 'Z', then a push-pull type of hardware driver will most likely be implemented. The technology library and the VHDL architectural determine the kind of output design.

A major problem of a *buffer* port is in the port mapping of component instantiations. The port mapping is restricted to either one (and only one) other port of mode *buffer* or to a signal of an architecture. If all the output ports throughout the hierarchy are of mode *buffer*, then there are no compilation problems. However, there are problems in connecting a *buffer* port to an *inout* or *out* port, or to more than one other *buffer* port. By careful use of intermediate signals, this problem is easily overcome. Figure 1.5.2-1 represents a component with buffer ports. Figures 1.5.2-2 and 1.5.2-3 demonstrate problem issues in using *buffer* ports.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity BufferN is
  port(A : in     Std_Logic_Vector(2 downto 0);
       B : buffer Std_Logic;
       C : buffer Std_Logic);
end BufferN;

architecture BufferN_a of BufferN is
begin
  B <= not A(2);           -- B is assigned
  C <= (A(1) and A(0)) or B; -- B is read in
end BufferN_a;
```

**Figure 1.5.2-1 Buffer Component (element\buffer2.vd)**

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity TOP2 is
  port(
    A : in     Std_Logic_Vector(2 downto 0);
    B : inout  Std_Logic;
    C : buffer Std_Logic
  );
end TOP2;

architecture TOP2_A of TOP2 is
  component BufferN
    port(
      A : in     Std_Logic_Vector(2 downto 0);
      B : buffer Std_Logic;
      C : buffer Std_Logic
    );
  end component;
```

If port mode is BUFFER, then instantiation BufferN\_i would operate correctly (See File element\buffertop.vhd )

```

begin
  BufferN_i: BufferN
  port map(
    A          => A,
    B          => B,
    C          => C
  );
end TOP2_A;

```

 Cannot associate BUFFER port B with INOUT port B.

**Figure 1.5.2-2 Instantiation of Component with Buffer Error  
(element\bufftop2.vhd)**

```

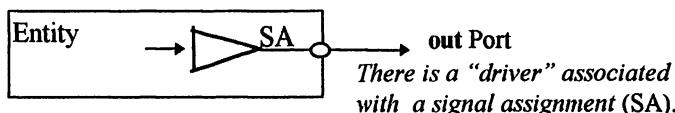
architecture TOP3_A of TOP2 is
...
signal B_s : Std_Logic;
begin
  BufferN_i: BufferN
  port map (
    A          => A,
    B          => B_s,
    C          => C);
  B <= B_s;
  B_s <= 'Z';
end TOP3_A;

```

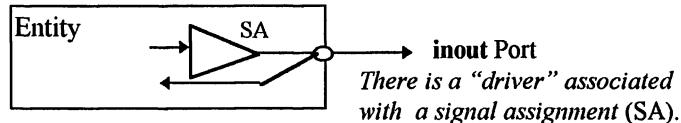
 Signal B\_s must have only one source since it is connected to a buffer port. Commenting this line fixes the problem.

**Figure 1.5.2-3 Instantiation of Component with Buffer Error  
(element\bufftop3.vhd)**

A port of mode *out* represents an output. Signal assignments can be made to an *out* port, but data from an *out* port cannot be read. If the value assigned to an *out* port needs to be read, then the user needs to define a signal within the architecture, and assign values to this architectural signal (as discussed above). A concurrent statement can then be used to assign the value of this signal onto the *out* port. This concept is demonstrated in figure 1.5.2-4. Another alternative is to use an *inout* port instead of an *out* port. However, it is important that the model reflects the true intent of the port, rather than to conform to any peculiarity of the VHDL. For example, a bi-directional processor data bus must be defined as an *inout* port, whereas a *Read* or a *Write* control signal sourced by a microprocessor should be defined as an *out* port.



A port of mode *inout* represents a bi-directional interface. Signal assignments can be made to an *inout* port, and data can be read from an *inout* port.



```
entity BufferAlternative is
  port(A      : in   Bit_Vector(2 downto 0);
        B      : out  Bit;
        C      : out  Bit);
end BufferAlternative;

architecture BufferAlternative_a of BufferAlternative is
  signal T_s : Bit;
begin
  T_s <= A(2) and A(1);
  B    <= T_s;
  C    <= T_s or A(0);
end BufferAlternative_a;
```

**Use out instead of buffer**

**T<sub>s</sub> signal is assigned to port B, and is read for the port C calculations.**

**Figure 1.5.2-4 Alternative use of *Buffer* Ports with *Out* Ports  
(element\buffer.vhd)**

### 1.4.3 Port Association Rules – the Open

**Q** | When is it legal to connect a port to an *open*?

**A** | Per LRM 1.1.1.2, [1] if a *formal port* is associated with an *actual port*, *signal*, or *expression* (VHDL'93 only), 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. This is equivalent to expressing that in a printed circuit board, if a pin of a device (*the formal of the component*) is wired (or associated) to a trace of the board (*the actual*), then that pin is connected to the printed circuit board. However, if that pin is associated to *open*, then that pin is unconnected.

[1] A port of mode *in* may be unconnected or unassociated only if its declaration includes a default expression. 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 ports are unconnected and others are either unconnected or unassociated. Figure 1.5.3 demonstrates the port association concepts. Section 7.24 provides a discussion of the port mapping association to *Vcc* (+5 Volts) or to *Ground*.

```

entity Open_Nty is
  port (InPlain      : in      integer;
        InInit       : in      integer := 0;
        InOutPlain   : inout   integer;
        InOutInit    : inout   integer := 10;
        BufferPlain  : buffer  integer;
        OutPlain     : out     integer;
        InBVU        : in      Bit_Vector; -- unconstrained array
        InOutBVU     : inout   Bit_Vector;
        InBVIInit   : in      Bit_Vector(7 downto 0) -- -- constrained array
                                         := X"AB"
      );
end Open_Nty;
entity OpenTB_Nty is -- Testbench
end OpenTB_Nty;

architecture OpenTB_A of OpenTB_Nty is
  signal TestBV_s : Bit_Vector(7 downto 0);
  signal Int_s : integer := 100;

component Open_Nty -- Component declaration
  port(
    InPlain      : in      integer;
    InInit       : in      integer := 0;
    InOutPlain   : inout   integer;
    InOutInit    : inout   integer := 10;
    BufferPlain  : buffer  integer;
    OutPlain     : out     integer;
    InBVU        : in      Bit_Vector;
    InOutBVU     : inout   Bit_Vector;
    InBVIInit   : in      Bit_Vector(7 downto 0)
                                         := X"AB"
  );
end component;

begin
  U1_Open_Nty: Open_Nty
    port map ( -- Component instantiations
      InPlain      => open, -- uninitialized IN port
      InInit       => open, -- OK per LRM 1.1.1.2
      InOutPlain   => open, -- OK per LRM 1.1.1.2
      InOutInit    => open, -- OK per LRM 1.1.1.2
      BufferPlain  => open, -- OK per LRM 1.1.1.2
      OutPlain     => open, -- OK per LRM 1.1.1.2
      InBVU        => open, -- Error per LRM 1.1.1.2
                           -- Unconstrained array
      InOutBVU     => open, -- Error per LRM 1.1.1.2
                           -- Unconstrained array
      InBVIInit   => open
    );
  U2_Open_Nty: Open_Nty
    port map (
      InPlain      => 20,   -- error for VHDL'87 only
                           -- 87
                           -- 93 (20 is an expression)
      InInit       => Int_s,
      InOutPlain   => Int_s,
      InOutInit    => open, -- OK per LRM 1.1.1.2
      BufferPlain  => open, -- OK per LRM 1.1.1.2
      OutPlain     => open, -- OK per LRM 1.1.1.2
      InBVU        => TestBV_s,
      InOutBVU     => TestBV_s(3 downto 0),
    );
  
```

Component Declaration.

Component instantiations

87

93 (20 is an expression)

```

-- Formal InBVIInit must not be associated with OPEN
-- when subelements are associated individually.

InBVIInit(7)          => open, -- error per LRM 1.1.1.2
InBVIInit(6 downto 0)  => TestBV_s(6 downto 0)
);
end OpenTB_A;

```

**Figure 1.5.3 Port Association Examples (element\open\_ea.vhd)****1.4.4 Type Conversion in Port Associations**

**Q** | How can ports of different types be connected in component instantiations?

**A** | When formal parameters of a port are of different type than their associated actual parameters, function calls or type conversions (for VHDL'93 only) can be used within the association list to convert to the correct types (see LRM 4.3.2.2). Thus, conversions are used only to ensure type matching. Table 1.5.4 summarizes the association list rules. Figures 1.5.4-1 and 1.5.4-2 represent examples of the conversion rules in port associations for VHDL'87 and VHDL'93.

**Table 1.5.4 Port Association List**

FORMAL mode	FORMAL Type Conversion	ACTUAL Type Conversion	COMMENTS
in	none	FActual_2_formal(ad)	Actual must NOT be open
inout	Fformal_2_Actual(fd)	FActual_2_formal(ad)	Actual must NOT be open
out	Fformal_2_Actual(fd)	none	Actual must NOT be open
buffer	Fformal_2_Actual(fd)	none	Actual must NOT be open
in	none	Tformal(ad)	Actual must NOT be open
inout	TActual(fd)	Tformal(ad)	Actual must NOT be open
out	TActual(fd)	none	Actual must NOT be open
buffer	TActual(fd)	none	Actual must NOT be open

Fformal\_2\_Actual(fd) = Type conversion function from the formal type to the actual type with the formal designator (fd) as the parameter -- VHDL'87 & VHDL'93

Factual\_2\_formal(ad) = Type conversion function from the actual type to the formal type with the actual designator (ad) as the parameter -- VHDL'87 & VHDL'93

Tactual(fd) = VHDL Type conversion using type mark of actual with the formal designator (fd) as the parameter -- **FOR VHDL'93 ONLY**

Tformal(ad) = VHDL Type conversion using type mark of formal with the actual designator (ad) as the parameter -- **FOR VHDL'93 ONLY**

```

entity Alist_Nty is
  port (InReal_p : in      Real;
        OutInt_p : out     Integer);
end Alist_Nty;

architecture Alist_a of Alist_Nty is
begin -- Alist_a
  -- Concurrent signal assignments
  OutInt_p <= integer(InReal_p); -- Type conversion, legal VHDL'87 &'93
end Alist_a;

-----
-- File name   : alisttb.vhd
-- Title       : Testbench for list association test
-- Description : Tests of various association methods
entity ListTB_Nty is
end ListTB_Nty;
architecture ListTB_A of ListTB_Nty is
  function ToReal(Integer_v : integer) return real is
    begin
      return real(integer_v);
    end ToReal;
  component Alist_Nty
    port( InReal_p          : in      real;
          OutInt_p         : out     integer);
  end component;

  signal Real_s   : real := 3.14;
  signal Real2_s  : real := 0.0;
  signal Int1_s   : integer := 1;
  signal Int2_s   : integer := 2;

begin
  U1_Alist_Nty: Alist_Nty
    port map(
      InReal_p      => Real_s,      -- formal real gets actual real
      OutInt_p      => Int1_s);    -- actual integer gets formal integer

  U2_Alist_Nty: Alist_Nty
    port map(           -- formal real gets converted actual integer
      InReal_p      => ToReal(Int2_s),
      ToReal(OutInt_p) => Real2_s); -- actual real gets converted formal integer

  U3_Alist_Nty: Alist_Nty
    port map(           -- Formal real gets converted actual integer
      InReal_p      => ToReal(Int2_s),
      --ToReal(OutInt_p) => open -- Illegal, conversion function with open
      OutInt_p      => open);
end ListTB_A;

```

**Figure 1.5.4-1 Examples of Port Association Conversion Rules, VHDL'87 & VHDL'93 (element\alist\_ea.vhd, element\alisttb.vhd)**

```

...
architecture ListTB_A of ListTB_Nty is
...
begin
...
U2_Alist_Nty: Alist_Nty
  port map (
    InReal_p      => Real(Int2_s), -- formal real gets converted actual integer
    Real(OutInt_p) => Real2_s   -- actual real gets converted formal integer
  );
U3_Alist_Nty: Alist_Nty
  port map (
    InReal_p      => Real(Int2_s), -- Formal real gets converted actual integer
    --ToReal(OutInt_p) => open -- Illegal, conversion function with open
    OutInt_p      => open
  );
end ListTB_A;

```

**Figure 1.5.4-2 Examples of Port Association Conversion Rules, VHDL'93  
(element\alisttb2.vhd)**

Section 8.7 presents another application of the port association conversion rule.

## 1.5 CONFIGURATIONS

### 1.5.1 Configuration Requirements

**Q** Is there a mandatory requirement for full configuration throughout a structural design? If one writes configuration declarations, must they be compiled into the same library as the relevant entity and architecture?

**A** There is no mandatory requirement for a configuration in the LRM. The LRM provides rules for the default binding of components. This avoids the need for configuration specifications and configuration declarations. It is possible to have a complete hierarchical design without a single configuration in it anywhere. Binding is defined as [1] *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.*

The architecture that instantiates the components must have access to the libraries where the components are compiled through the use of the *library <Library\_Name>* statements. Some VHDL systems interpret the LRM to mean that all entities in library *WORK* are always visible, so default binding of entities in *WORK* happens automatically. Other systems interpret the LRM to mean that no entities are implicitly visible, and a *use* clause will be necessary for library *WORK*. The latter interpretation is probably the correct one, but the former is the more common one.

Some simulators require a configuration specification or declaration. A configuration declaration is a design unit and can be compiled into any library.

### 1.5.2 Configuration Specification/Declaration

**Q** Which is recommended, a configuration specification or configuration declaration? Can examples of each be provided?

**A** The explicit binding of components is often used in VHDL to examine the modeling of a different representation of the design under test. In a top-down design

methodology, these representations could include the modeling of a unit under test (UUT) at various levels of designs. These levels include behavioral level, register transfer Level (RTL) prior to synthesis, or structural level with representations and connection of every gate and flip-flop in the design. Note that some synthesizers can automatically write synthesized designs into structural VHDL representation, but it is only machine readable (i.e., it is not for humans). There are three ways to perform binding:

1. **Configuration specification:** This approach is used to specify the bindings of components in the architecture that instantiates these components
2. **Configuration declaration:** This approach uses a separate design unit to specify the binding of components to entities and architectures.
3. **Default Binding Indication:** The default binding for a component occurs for an architecture where no binding indication is provided, and where the component identifier is the same as the entity name. The binded architecture is **the most recently analyzed architecture body** associated with the entity declaration. The architecture identifier is determined during elaboration. The default binding indication includes a default generic map defined in the entity declaration.

**M** Whenever possible, when declaring components in an architecture or in a package, use as the component name the entity associated with that component.

**Rationale:** *This approach enhances code readability because the name of the entity associated with the component is the identifier for the component. In addition, it allows the use of default binding for simple design cases. Some synthesizers require that the default binding be used.*

**M** Use the configuration declaration method to bind components to entities and architectures if more than one architecture exists for the entity.

**Rationale:** *Configuration declarations represent separate design units and allow for late binding of components.*

#### 1.5.2.1 Configuration Specification

[1] A configuration specification associates binding information with component labels representing instances of a given component declaration. Thus, if an architecture instantiates a

component then that architecture can specify a binding for each instance of that component to a particular architecture. The component may have several architectural representations (e.g., behavior, gate level). [1] A *binding indication* associates component instances with a particular design entity. If an explicit binding indication is not provided, then the default binding indication will apply. Figure 1.6.2.1 represents a configuration specification example with components of the same and different names as their corresponding entity names. Example of the default configuration specification is also demonstrated.

```

entity AndGate_Nty is
  generic (DlyIO_g: time := 4 ns);
  port (I1: in Bit;
        I2: in Bit;
        O:   out Bit);
end AndGate_Nty;

entity XorGate_Nty is
  generic (DlyIO_g: time := 4 ns);
  port (I1: in Bit;
        I2: in Bit;
        O:   out Bit);
end XorGate_Nty;
-----
```

Compile the examples  
in this section in  
library *Atep.Lib*

```

architecture AndGate_a of AndGate_Nty is
begin -- AndGate_a
  O <= I1 and I2 after DlyIO_g;
end AndGate_a;
-----
```

```

architecture XorGate_a of XorGate_Nty is
begin -- XorGate_a
  O <= I1 xor I2 after DlyIO_g;
end XorGate_a;
-----
```

```

package Gates_Pkg is
  component AndGate
    generic (Dly_g: time := 4 ns);
    port (A : in Bit;
          B : in Bit;
          C : out Bit);
  end component;
  component AndGate_Nty
    generic (DlyIO_g: time := 4 ns);
    port (I1 : in Bit;
          I2 : in Bit;
          O : out Bit);
  end component;
  component XorGate_Nty
    generic (DlyIO_g: time := 4 ns);
    port (I1 : in Bit;
          I2 : in Bit;
          O : out Bit);
  end component;
end Gates_Pkg;
-----
```

This component has a different  
name and different interfaces than  
the entity AndGate\_Nty

```

library ATEP_Lib;
use ATEP_Lib.Gates_Pkg.all;

entity Structure_Nty is
end Structure_Nty;

architecture Structure_a of Structure_Nty is
-- This architecture implements the equation:
-- Out_s <= ((In1_s and In2_s) and In3_s) xor In4_s;
signal      And2a_s          : Bit := '0';
signal      And2b_s          : Bit := '0';
signal      In1_s            : Bit := '1';
signal      In2_s            : Bit := '1';
signal      In3_s            : Bit := '1';
signal      In4_s            : Bit := '0';
signal      Out_s            : Bit := '0';

-- This example demonstrates the mapping of generics and ports
-- that have different names between the component AndGate and
-- its corresponding mapping
-- to AndGate_Nty
for U1_AndGate: AndGate use
  entity ATEP_Lib.AndGate_Nty(AndGate_a)
    generic map(DlyIO_g => Dly_g)
    port map
      (I1 => A,
       I2 => B,
       O => C);
  -- Since the name of the component is the same as its
  -- corresponding entity, no port map is required.
  for U2_AndGate_Nty: AndGate_Nty use
    entity ATEP_Lib.AndGate_Nty(AndGate_a);

    -- Since NO binding is defined for XorGate_Nty, the default bind is
    -- used (i.e. XorGate_Nty entity and latest compiled XOR architecture
    -- for this entity
begin -- Structure_a
  U1_AndGate: AndGate
    generic map(Dly_g => 5 ns)           -- component instantiations
    port map
      (A  => In1_s,
       B  => In2_s,
       C  => And2a_s); -- (In1_s and In2_s )

  U2_AndGate_Nty: AndGate_Nty
    -- generic map(DlyIO_g => 5 ns)
    port map
      (I1 => And2a_s,
       I2 => In3_s,
       O  => And2b_s); -- ((In1_s and In2_s) and In3_s)

  U3_XorGate_Nty: XorGate_Nty
    generic map(DlyIO_g => 6 ns)
    port map
      (I1 => And2b_s,
       I2 => In4_s,
       O  => Out_s); -- ((In1_s and In2_s) and In3_s) xor In4_s;
end Structure_a;

```

**Actual component name is "AndGate"**  
that is mapped into "AndGate\_Nty"

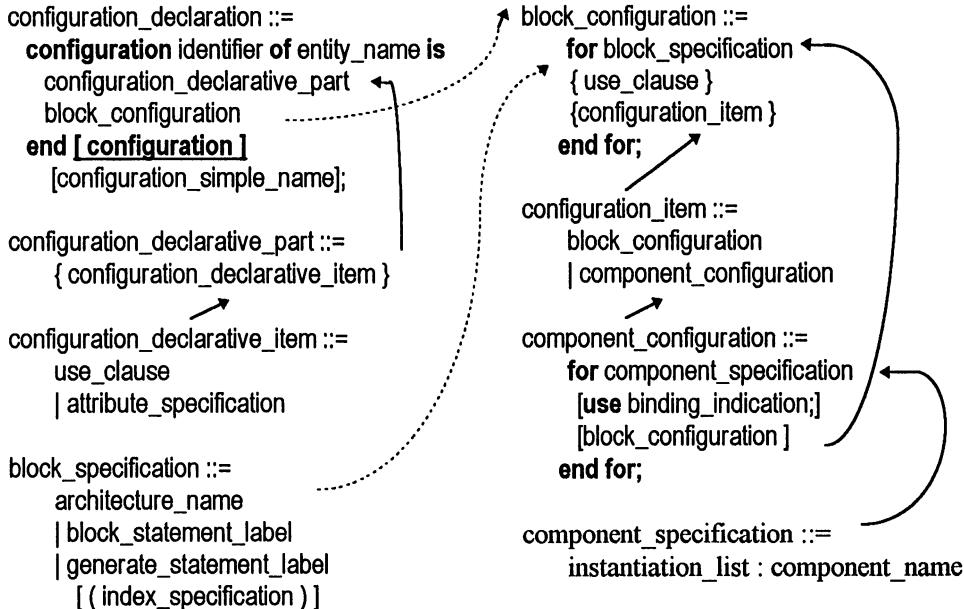
**Formal parameters of AndGate\_Nty mapped (=>) to**  
**Actual parameters of AndGate component**

**The generic can be omitted if default**  
**is intended, or if configuration**  
**declaration redefines it.**

Figure 1.6.2.1 Configuration Specification Example (element\cfgspec.vhd)

### 1.5.2.2 Configuration Declaration

The configuration declaration uses a separate design to specify the binding of components to entities and architectures. The syntax is as follows:



Given the same packages and components used in the previous example, file *cfgdecl.vhd* was modified to reflect the deferred binding of components in architecture *structure\_a*. In addition, a configuration design unit was added to show the late binding. Figure 1.6.2.2-1 demonstrates the *structure\_a* architecture with the deferred binding, and the configuration design unit.

```

library ATEP_Lib;
use ATEP_Lib.Gates_Pkg.all;

entity Structure_Nty is
end Structure_Nty;

architecture Structure_a of Structure_Nty is
    signal And2a_s      : Bit := '0';
    signal And2b_s      : Bit := '0';
    signal In1_s        : Bit := '1';
    signal In2_s        : Bit := '1';
    signal In3_s        : Bit := '1';
    signal In4_s        : Bit := '0';
    signal Out_s         : Bit := '0';

```

– This architecture implements the equation:  
–  $Out\_s \leq ((In1\_s \text{ and } In2\_s) \text{ and } In3\_s) \text{ xor } In4\_s;$   
Compile the examples in this section in library  
*Atep.Lib*

```

begin -- Structure_a
  U1_AndGate: AndGate
    generic map(Dly_g => 5 ns)
    port map
      (A => In1_s,
       B => In2_s,
       C => And2a_s); -- (In1_s and In2_s)
  U2_AndGate_Nty: AndGate_Nty
    -- generic map(DlyIO_g => 5 ns) ←
    port map
      (I1 => And2a_s,
       I2 => In3_s,
       O => And2b_s); -- ((In1_s and In2_s) and In3_s)
  U3_XorGate_Nty: XorGate_Nty
    generic map(DlyIO_g => 6 ns)
    port map
      (I1 => And2b_s,
       I2 => In4_s,
       O => Out_s); -- ((In1_s and In2_s) xor In4_s);
end Structure_a;

library ATEP_Lib; -- Library where components are compiled
configuration Structure_Cfg of Structure_Nty is
  for Structure_a -- Top level architecture of Structure_Nty
    for U1_AndGate: AndGate use
      entity ATEP_Lib.AndGate_Nty(AndGate_a) ←
        generic map(DlyIO_g => Dly_g)
        port map
          (I1 => A, ← Formal parameters of AndGate_Nty mapped (=>) to
           I2 => B, ← Actual component name is "AndGate"
           O => C); Actual parameters of AndGate component
    end for;
    -- Since the name of the component is the same as its
    -- corresponding entity, this statement binds the entity
    -- to the desired architecture
    for U2_AndGate_Nty: AndGate_Nty use
      entity ATEP_Lib.AndGate_Nty(AndGate_a);
    end for;
    -- Since NO binding is defined for XorGate_Nty, the default bind is
    -- used (i.e. XorGate_Nty entity and latest compiled XOR architecture for this entity.
    -- ALL libraries which are visible will be searched (e.g. WORK, ATEP_LIB)
  end for;
end Structure_Cfg;

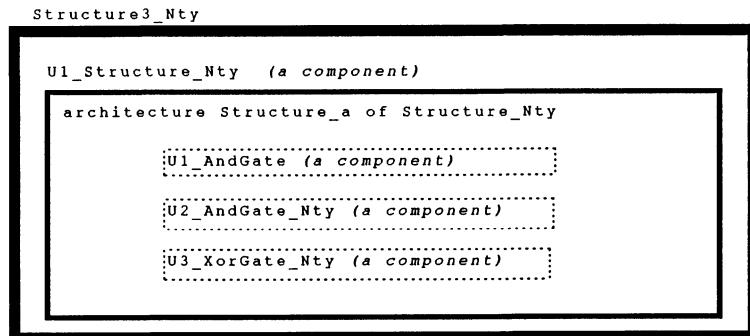
```

**Figure 1.6.2.2-1 Architecture and Configuration design unit  
(element\cfgdecl.vhd)**

An example of a hierarchical design is shown graphically in Figure 1.6.2.2-2 An example, binding hierarchical components in a configuration declaration, is shown in figure 1.6.2.2-3.

A configuration declaration overrides a configuration specification. However, it is generally preferred to only use the **configuration declaration** method to bind components to entities and architectures if more than one architecture exists for the entity. The

configuration declarations represent separate design units and allow for late binding of components.



**Figure 1.6.2.2-2 Hierarchical Design**

```

... -- File includes all the sublevels
entity Structure3_Nty is
end Structure3_Nty;

architecture Structure3_a of Structure3_Nty is
  component Structure_Nty
  end component;

begin
  U1_Structure_Nty: Structure_Nty; -- component instantiation
end Structure3_a;
-----
library ATEP_Lib;
use ATEP_Lib.Gates_Pkg.all;

configuration Structure3_Cfg of Structure3_Nty is
  for Structure3_a
    for U1_Structure_Nty: Structure_Nty           -- struct3_a
      for Structure_a                            -- U1_Struct
        for U1_AndGate: AndGate use              -- Struct_a
          entity ATEP_Lib.AndGate_Nty(AndGate_a )
            generic map(DlyIO_g => Dly_g)
            port map
              (I1 => A,
               I2 => B,
               O  => C);
        end for;                                -- U1_And
        for U2_AndGate_Nty: AndGate_Nty use       -- U2_And
          entity ATEP_Lib.AndGate_Nty(AndGate_a );
        end for;                                -- U2_And
        for U3_XorGate_Nty: XorGate_Nty  use      -- U3_Xor
          entity ATEP_Lib.XorGate_Nty(XorGate_a );
        end for;                                -- U3_Xor
      end for;                                -- Struct_a
    end for;                                -- U1_Struct
  end for;                                -- struct3_a
end Structure3_Cfg;

```

**Figure 1.6.2.2-3 Binding of Components of a hierarchical design with a Configuration Declaration (element\hierarch.vhd)**

### 1.5.2.3 Binding with configured components

If an architecture instantiates a component that already has a configuration defined, then that configuration can make use of that previously defined configuration for that component. Figure 1.6.2.3 represents an example that instantiates the component Structure\_Nty. This method allows for the configuration of hierarchical designs.

```
library ATEP_Lib;
configuration Structure3_Cfg of Structure3_Nty is
  for Structure3_a
    for U1_Structure_Nty: Structure_Nty
      use configuration ATEP_Lib.Structure_Cfg;
    end for;
  end for;
end Structure3_Cfg;
```

**Figure 1.6.2.3 Binding with Configured Component (element\struct3.vhd)**

### 1.5.2.4 Removing a Component from an Architecture through Configuration

Per LRM 12.4.3, *elaboration of 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 is bound to a configuration of such design entity.* Thus, the open binding can be used to effectively remove a component from an architecture without making modifications to the architecture that instantiates this component.

Figure 1.6.2.4 represents a configuration that effectively removes a component from an architecture. This feature did not operate correctly on some simulators. The user is cautioned to experiment with the simulator of choice(s) and to contact the vendor who is not compliant to the LRM. A second option to effectively disconnect pins of a component from their interconnected signals, is to use a separate architecture with the component commented out or removed. A third option is to use the error injector, described in section 6.3, to force the connections of the desired component interfaces to 'Z's.

```
library ATEP_Lib;
configuration Structure2_Cfg of Structure_Nty is
  use ATEP_Lib.Gates_Pkg.all;
  for Structure_a -- Top level architecture of Structure_Nty
    for U1_AndGate: AndGate use
      entity ATEP_Lib.AndGate_Nty(AndGate_a)
        generic map(DlyIO_g => Dly_g)
        port map
          (I1 => A,
           I2 => B,
           O  => C);
    end for;
```

```

for U2_AndGate_Nty: AndGate_Nty use
    entity ATEP_Lib.AndGate_Nty(AndGate_a);
end for;

-- U3_XorGate_Nty is effectively removed from the model.
for U3_XorGate_Nty: XorGate_Nty use open;
end for;
end for;
end Structure2_Cfg;

```

A component may be removed to emulate either a board level option (e.g. co-processor) or a fault condition.

**Figure 1.6.2.4 Configuration to Remove a Component from an Architecture  
(element\deferd\_c.vhd)**

### 1.5.3 Configuration Statement for a Specific Generic

**Q** Using an  $n$  stage filter, how can a configuration specify one filter as three stages, and another filter as four stages? Can this be done within a configuration by setting the generic values?

**A** Yes, a configuration declaration can be used to map generics on different instances of the filter. Figure 1.6.3 represents a dummy model of a filter where the different values of the generic are passed to the different stages. The package *image\_pkg* (described in section 5.1) is used to convert typed objects into strings.

```

library Work;
use Work.Image_Pkg.all;
entity Filter is
    generic (Stage_g : Natural := 3);
    port(A : in Bit_Vector(7 downto 0);
         B : out Bit_Vector(7 downto 0));
end Filter;

architecture Filter_a of Filter is
begin -- Filter_a
    assert False
        report "Stage = " & Work.Image_Pkg.Image(Stage_g) &
               " A = " & Work.Image_Pkg.Image(A) &
               " B'length = " & Work.Image_Pkg.Image(B'length)
        severity Note;
end Filter_a;
entity FilterStructure is
end FilterStructure;

architecture FilterStructure_a of FilterStructure is
component Filter
    generic (Stage_g : Natural := 3);
    port(
        A : in Bit_Vector(7 downto 0);
        B : out Bit_Vector(7 downto 0));
end component;

```

```

signal A1 : Bit_Vector(7 downto 0);
signal B1 : Bit_Vector(7 downto 0);
signal A2 : Bit_Vector(7 downto 0);
signal B2 : Bit_Vector(7 downto 0);
signal A3 : Bit_Vector(7 downto 0);
signal B3 : Bit_Vector(7 downto 0);
begin -- FilterStructure_a
    U1_Filter: Filter
        generic map(Stage_g => 5)
        port map(
            A          => A1,
            B          => B1);

    U2_Filter: Filter
        generic map(Stage_g => 5)
        port map(
            A          => A2,
            B          => B2);

    U3_Filter: Filter
        generic map(Stage_g => 5)
        port map(
            A          => A3,
            B          => B3);
end FilterStructure_a;

configuration Filter_Cfg of FilterStructure is
    for FilterStructure_a
        for U1_Filter: Filter use entity Work.Filter(Filter_a)
            generic map(Stage_g => 1);
        end for;

        for U2_Filter: Filter use entity Work.Filter(Filter_a)
            generic map(Stage_g => 2);           ← Setting the generics
        end for;

        for others: Filter use entity Work.Filter(Filter_a)
            generic map(Stage_g => 3);
        end for;
    end for;
end Filter_Cfg;

```

The diagram illustrates the code structure. It shows three instances of the 'Filter' component: U1\_Filter, U2\_Filter, and U3\_Filter. Each instance has its own generic mapping (Stage\_g => 1, 2, or 3 respectively) and port mappings (A to A1/A2/A3, B to B1/B2/B3). To the right of the instantiation section, a callout box labeled 'Multiple instantiations of filter component' points to the three 'Filter' declarations. Below the configuration section, another callout box labeled 'Setting the generics' points to the 'generic map' statements within the 'for' loops.

**Figure 1.6.3 Setting Generics with Configuration Declarations  
(element/filter2.vhd)**

#### 1.5.4 CONFIGURATION OF GENERATE STATEMENTS

**Q** | How can a configuration be defined for components instantiated inside a generate statement?

**A** | The previous section discussed the syntax for a configuration declaration. Figure 1.6.4-1 demonstrates the application of that syntax for the configuration declaration of components instantiated inside a generate statement. A parity tree is used as an example, with two architectures for the exclusive OR component.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
```

```
entity XOR_Nty is
port(A : in Std_Logic;
      B : in Std_Logic;
      Z : out Std_Logic);
```

```
end XOR_Nty;
```

**Two architectures for same entity.**

```
architecture XOR_a of XOR_Nty is
begin -- XOR_a
  Z <= A xor B;
end XOR_a;

architecture XOR2_a of XOR_Nty is
begin -- XOR_a
  Z <= ((not A) and B) or (A and (not B));
end XOR2_a;
```

```
library IEEE;
use IEEE.Std_Logic_1164.all;
```

```
entity Generate_Nty is
port(Ain1 : in Std_Logic_Vector(31 downto 0);
      Aout : out Std_Logic_Vector(31 downto 0));
```

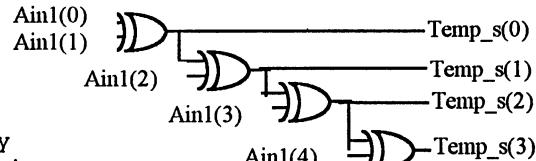
```
end Generate_Nty;

architecture Generate_a of Generate_Nty is
component XOR_Nty
port(A : in Std_Logic;
      B : in Std_Logic;
      Z : out Std_Logic);
```

```
end component;
```

```
signal Temp_s : Std_Logic_Vector(31 downto 0);
```

```
begin -- Generate_a
  Aout <= Temp_s;
```



-- Parity tree, 5 Bit parity

UK : for K\_i in 0 to 3 generate

UK0 : if K\_i = 0 generate

UXOR : XOR\_Nty

```
      port map(A => Ain1(K_i),
                 B => Ain1(K_i + 1),
                 Z => Temp_s(K_i));
```

end generate UK0;

UK1\_3 : if K\_i > 0 generate

UXOR : XOR\_Nty

```
      port map(A => Temp_s(K_i - 1),
                 B => Ain1(K_i + 1),
                 Z => Temp_s(K_i));
```

end generate UK1\_3;

end generate UK;

```
end Generate_a;
```

**These generate statements create the exclusive OR tree shown above**

```

library Work;

configuration Parity_Cfg of Generate_Nty is
    for Generate_a
        for UK(0)
            for UK0
                for UXOR : XOR_Nty use
                    entity Work.XOR_Nty(XOR_a);
                end for;
            end for;
        end for;
    end for;

    for UK(1 to 2)
        for UK1_3
            for UXOR : XOR_Nty use
                entity Work.XOR_Nty(XOR2_a);
            end for;
        end for;
    end for;

    for UK(3)
        for UK1_3
            for UXOR : XOR_Nty use
                entity Work.XOR_Nty(XOR_a);
            end for;
        end for;
    end for;
end Parity_Cfg;

```

Figure 1.6.4-1 Configuration of Generate Statement (element\genrat\_c.vhd)

Figure 1.6.4-2 represents the configuration declaration of a hierarchical design with *generate* statements.

```

entity Multiplier is
    generic (A_g : Integer;
             B_g : Integer;
             C_g : Integer);
end Multiplier;

architecture Multiplier_a of Multiplier is
begin
end Multiplier_a;
-----
entity EEPROM is
    generic (A_g : Integer;
             B_g : Integer;
             C_g : String);
end EEPROM;

architecture EEPROM_a of EEPROM is
begin
end EEPROM_a;
-----
entity Slice is
end Slice;

```

```

architecture Slice_a of Slice is
component Multiplier
  generic (A_g : Integer;
           B_g : Integer;
           C_g : Integer);
end component;

begin
  mult_i : Multiplier
    generic map (A_g => 0,
                 B_g => 1,
                 C_g => 2);
end Slice_a;
=====
entity UUT is
end UUT;

architecture UUT_a of UUT is
  component Slice
  end component;

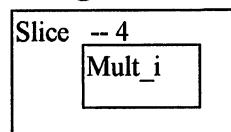
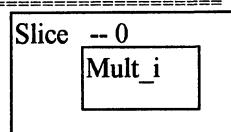
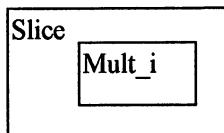
begin
  Gen_Lbl : for K in 0 to 4 generate
    Slice_i : Slice;
  end generate Gen_Lbl;
end UUT_a;
=====

entity TestBench is
end TestBench;

architecture TestBench_a of TestBench is
  component UUT
  end component;

  component EEPROM
    generic (A_g : Integer;
             B_g : Integer;
             C_g : String);
  end component;
begin -- TestBench_a
  UUT_i : UUT;

  EEPROM_i : EEPROM
    generic map(A_g => 0,
                B_g => 2,
                C_g => "ABC.TXT");
end TestBench_a;
=====
```



```

configuration Top_Cfg of TestBench is
    for TestBench_a                      -- testbench architecture
        for UUT_i : UUT                  -- UUT instance, UUT component
            use entity work.UUT(UUT_a);   -- Binding Indication, entity(architecture)
                for UUT_a                 -- architecture
                    for Gen_Lbl(0 to 3)      -- Generate label and indices
                        for Slice_i : Slice  -- Slice instance, Slice component
                            use entity WORK.Slice(Slice_a); -- Binding Indication, entity(architecture)
                                for Slice_a          -- slice architecture
                                    for mult_i : Multiplier -- instance, component
                                        use entity work.Multiplication(Multiplier_a) -- entity, architecture
                                            generic map(
                                                A_g => 18,
                                                B_g => 16,
                                                C_g => 33
                                            );
                                        end for;
                                    end for;
                                end for;
                            end for;
                        end for;
                    end for;
                end for;
            end for;
        end for;

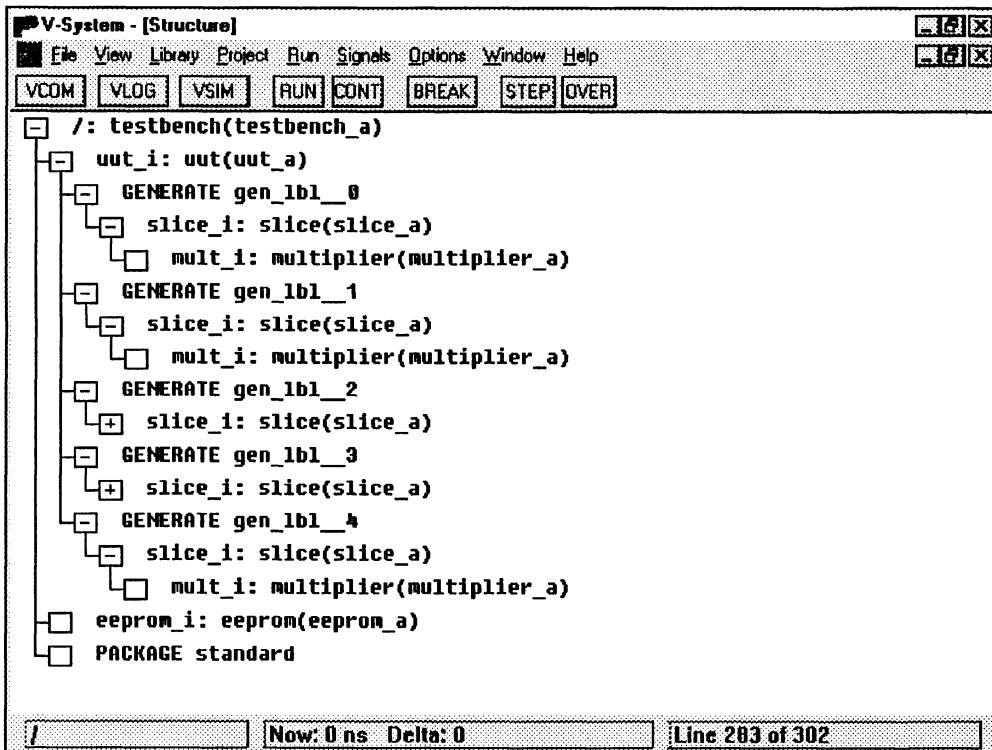
        for Gen_Lbl(4)                      -- Separate generate label index
            for Slice_i : Slice
                use entity WORK.Slice(Slice_a);
                    for Slice_a
                        for mult_i : Multiplier
                            use entity work.Multiplication(Multiplier_a)
                                generic map(
                                    A_g => 18,
                                    B_g => 16,
                                    C_g => 33
                                );
                            end for;
                        end for;
                    end for;
                end for;
            end for;
        end for;

        for EEPROM_i : EEPROM
            use entity work.EEPROM(EEPROM_a)
                generic map(
                    A_g => 128,
                    B_g => 2048,
                    C_g => "ABC.txt"
                );
        end for;
    end for;
end Top_Cfg;

```

**Figure 1.6.4-2 Configuration Declaration of a Hierarchical Design with generate statements (Element\topcfg.vhd)**

Figure 1.6.4-3 represents the structure of this design, as shown by Model Technology's *VSystem Structure* view.



**Figure 1.6.4-3 Structure of Design, as shown by *Vsystem Structure* view.**

## 1.6 ARITHMETIC ISSUES AND OPERATORS

### 1.6.1 Defining 2'S Complement Value -(2\*\*31) in VHDL

**Q** For behavioral simulation, it is often desired to use integers instead of *Std\_uLogic\_Vector* type. There are no problems with vectors up to 31 bits. VHDL integer type range is defined as  $-(2^{**}31 - 1)$  to  $(2^{**}31 - 1)$ , therefore the 2's complement value  $-(2^{**}31)$  is missing. This means that the 32-bit value '*T*' & (*others* => '0') cannot be represented by a standard VHDL integer. What is the best way to overcome the integer range limitations?

**A** LRM 3.1.2 specifies that an implementation must allow the declaration of any integer type whose range is wholly contained within the bounds -2147483647 and +2147483647 inclusive, or  $-(2^{**}31 - 1)$  to  $+(2^{**}31 - 1)$ . VHDL defines this range as the minimum range. It is essential to write portable code, and to restrict a model to this restricted range, even though some tools support extended ranges.

To declare a vector longer than 31 bits, the use of bit vector types is recommended. These types include *Bit\_Vector*, *Std\_Logic\_Vector*, *Std\_uLogic\_Vector*. Types *Signed* and *Unsigned* (defined in *Numeric\_Std*, *Numeric\_Bit*, or *Std\_Logic\_Arith* packages) are useful when arithmetic and multiplication operators are required.

### 1.6.2 Value Casting

**Q** | How can type casting be done in VHDL? In the C language, a value is casted as follows: (double) Integer\_Variable

**A** | Conversion functions for closely related types are implicitly declared (per LRM 7.3.5). In VHDL closely related types can be converted in a type casting fashion. All integer and real types are closely related. Closely related array types have the same elements, same number of dimensions, and the index types are closely related. One could not convert between *string* and *Bit\_Vector* because the elements are different (*Character* and *Bit* respectively). However, it is possible to use type conversions between different array types that have the same element type, like *Std\_Logic\_Vector* and *Std\_uLogic\_Vector* or *Unsigned* or *Signed*. The conversion format is: *TYPE(VALUE)*. Notice that the tick mark is absent; it represents a type qualifier rather than a type conversion. Examples of type conversions are show in figure 1.7.2.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;
entity Conversion is
  port (A : in Unsigned(3 downto 0);
        B : inout Std_Logic_Vector(3 downto 0));
end Conversion;

architecture Conversion_a of Conversion is
  type Short_Typ is range -128 to 127;
  type Medium_Typ is range -32768 to 32767;
  subtype BetterShort_Typ is Integer range -128 to 127;
  subtype BetterMedium_Typ is Integer range -32768 to 32767;

  signal S          : Short_Typ;
  signal M          : Medium_Typ;
  signal BetterS    : BetterShort_Typ;
  signal BetterM    : BetterMedium_Typ;
  signal Std_s      : Std_Logic_Vector(3 downto 0);
  signal Unsg_s     : Unsigned(3 downto 0);

begin
  Test_Lbl: process
    variable R_v : Real := 1.25;
    variable I_v : Integer := 4;
  begin
    R_v := Real(I_v) + 5.2;
    I_v := Integer(R_v);
    BetterM <= BetterS; -- Subtypes of parent type (integer)
    -- M <= S; -- Illegal Different types
    M <= Medium_Typ(S); -- Type conversion
    -- Std_s <= A; -- Different types
    Std_s <= Std_Logic_Vector(A); -- Type conversion
  end;
end;

```

```

-- Unsg_s <= Std_s; -- 
Unsg_s <= Unsigned(Std_s);
assert false
    report String'("1011");
    severity note;
end Test_Lbl;
end conversion_a;

```

Do not confuse implicit type conversion with type qualification, used to resolve ambiguities in the typing rules of VHDL. For example, a quoted string value could also be a Bit\_Vector.

**Figure 1.7.2 Examples of type conversions (element\convn.vhd)**

### 1.6.3 REM/MOD Difference

**Q**

What are the differences between the *mod* and *rem* operators?

**A**

*Mod* is the modulus operator. The *rem* is the remainder after division operator.  
 Note that  $X \text{ mod } Y = X \text{ rem } Y$  when  $\text{sign}(X) = \text{sign}(Y)$  (and  $Y \neq 0$ ).

However, when  $\text{sign}(X) \neq \text{sign}(Y)$ , the results may be different. The *Mod* operator is synthesizable, provided the modulus factor is a power of 2. Therefore:

$\text{Sig1} \leqslant X \text{ mod } 64$ ; -- is synthesizable

$\text{Sig2} \leqslant X \text{ mod } 63$ ; -- is non-synthesizable

Figure 1.7.3-1 provides an example of the *mod* and *rem* operators.

J	K	J mod K	J rem K	J	K	J mod K	J rem K
0	5	0	0	0	-5	0	0
1	5	1	1	1	-5	-1	1
2	5	2	2	2	-5	-2	2
3	5	3	3	3	-5	-3	3
4	5	4	4	4	-5	-4	4
5	5	0	0	5	-5	-0	0
				0	-5	0	0
0	5	0	0	-1	-5	-1	-1
-1	5	4	-1	-2	-5	-2	-2
-2	5	3	-2	-3	-5	-3	-3
-3	5	2	-3	-4	-5	-4	-4
-4	5	1	-4	-5	-5	-0	-0
-5	5	0	0				

**Figure 1.7.3-1 Example of the Mod and Rem Operators**

*Mod* is useful when defining counters (up or down) that roll over, or when computing a pseudo-random number within a desired range. A pseudo-random number generator making use of the *mod* operator is provided in section 5.5. Figure 1.7.3-2 demonstrates the use of the *mod* operator in a countdown counter. For example when the current value of *Count\_s* is 0, the next value is 3 because it is the result of  $-1 \text{ mod } 4$ .

```

entity Counter is
  generic
    (Modulus_g : Integer := 4);
  port
    (Count      : out Integer range 0 to Modulus_g - 1;
     Clock      : in Bit);
end Counter;

architecture Counter_Beh of Counter is
  signal Count_s : Integer range 0 to Modulus_g - 1;
begin
  Counter_Lbl: process
  begin
    wait until Clock'event and Clock = '1';
    -- Count(present)   Count(Next)
    --      0           3
    --      3           2
    --      2           1
    --      1           0
    Count_s <= (Count_s - 1) mod Modulus_g;
  end process Counter_Lbl;

  Count <= Count_s;  -- Concurrent statement
end Counter_Beh;

```

**Figure 1.7.3-2 Mod operator for a Count-Down Counter  
(element\count\_ea.vhd)**

#### 1.6.4 Non-Associative Operators

**Q**

Is it illegal to combine two or more *NANDs* or *NORs*? For example:

$F \leq A \text{ nand } B \text{ nand } C;$

Why not let the left-to-right rule cause the meaning to be:

$F \leq (A \text{ nand } B) \text{ nand } C; \dots ?$

**A**

For non-associative operators such as *NANDs* and *NORs*, parentheses are required. One might forget the left-to-right rule and assume that  $A \text{ nand } B \text{ nand } C$  equals  $A \text{ nand } (B \text{ nand } C)$ , which it does not. However, since  $(A \text{ and } B) \text{ and } C = A \text{ and } (B \text{ and } C)$ , it is allowed to write  $A \text{ and } B \text{ and } C$ . Figure 1.7.4 demonstrates the use of non-associative operators. Without the parentheses, the code will fail to compile.

```

...
architecture Test_a of Test is
  signal A, B, C, F, G, H, K, M : bit;
begin
  F <= A nand (B nand C);  -- Parentheses are required
  G <= A nand B nand C;  -- ERROR ⚡
  H <= A and B and C;
  K <= A nor B nor C;    -- ERROR ⚡
  M <= A or B or C;
end Test_a;

```

**Figure 1.7.4 Non-Associative Operators (element\left2r.vhd)**

### 1.6.5 Analysis Error with the “&” Operator

**Q**

Why does the code shown in figure 1.7.5-1 yield an analysis error, and how can it be corrected?

```
architecture CaseE_a of CaseE is
begin
  CaseE_Lbl: process(A, B, C) -- A, B, C are "in" ports
  begin
    case A & B is      -- ERROR ⚡*
      when "00" => Q <= C;
      when "01" => Q <= A;
      when "10" => Q <= B;
      when others => Q <= '0';
    end case;
  end process CaseE_Lbl;
end CaseE_a;
```

Figure 1.7.5-1 Error in *case* Expression (synths\concat.vhd)

**A**

The problem is that the “&” operator yields an unconstrained vector, and the error message is *Array case expression must have a static subtype*. Figure 1.7.5-2 provides three solutions.

```
CaseE_Lbl: process(A, B, C)
begin
  case Bit_Vector'(A, B) is
    when "00" => Q <= C;
    ...
  end case;
end process CaseE_Lbl;

CaseE_Lbl: process(A, B, C)
  subtype BV2_Typ is Bit_Vector(1 to 2);
begin
  case BV2_Typ'(A & B) is
    when "00" => Q <= C;
    ...
  end case;
end process CaseE_Lbl;

CaseE_Lbl: process(A, B, C)
  variable BV2 : Bit_Vector(1 to 2);
begin
  BV2 := A & B;
  case BV2 is
    when "00" => Q <= C;
    ...
  end case;
end process CaseE_Lbl;
```

Use of a type qualifier for an aggregate. The array is constrained by the size of the aggregate, just like the following constant declaration:  
**constant X : Bit\_Vector := "1010";**

Use of a type qualifier for an expression

Use of a variable

Figure 1.7.5-2 Corrections for Error in *case* Expression (element\concat.vhd)

### **1.6.6 Specifying a Multi-Line String.**

**Q** For a one-hot enumeration encoding scheme it is desired to define a VERY long string. How can this be defined using multiple lines? For a one-hot encoding, the Synopsys® manuals suggest the following syntax (for a small FSM):

```

type State_Typ is (S0, S1, S2, S3); -- 4 states, 4 flip-flops
attribute ENUM_ENCODING : String;
attribute ENUM_ENCODING of State_Typ: type
    is "0001 0010 0100 1000"; -- string length = 4**2 + 3

```

If the FSM has 66 states, the length of the state register is 66 bits, or one flip-flop per machine state. The length of the string for the attribute ENUM\_ENCODING is  $66**2$  character (definition of all possible enumeration codes) + 65 space characters, or 4356 + 65 (4421 characters total).

A

The standard string concatenation could be used as follows:

1.7 PACKAGE STD LOGIC 1164

### 1.7.1 Differences Std Logic 1164'1987 and '1993

**Q** Figure 1.8.1 demonstrates a package that makes use of *Std\_Logic\_1164* conversion, but yields errors when compiled with the VHDL'87 version of the Model Technology Version 4.2f compiler. What are the causes of these errors?

```
library IEEE;
use IEEE.Std_Logic_1164.all;
package Confused is
    constant Z1 : Std_uLogic_Vector(7 downto 0) :=  
        To_StdULogicVector(X"E4");
-- VHDL'87 Works fine. ☺  

-- VHDL'93 yields the following error: !  

--   "To_StdULogicVector" is ambiguous,  

--   Type error resolving function call: "To_StdULogicVector".  

constant Z2 : Std_uLogic_Vector(7 downto 0) :=  
    To_StdULogicVector(Bit_Vector(X"E4"));
-- VHDL'87 works fine ☺  

-- VHDL'93 works fine ☺
```

```

constant Z3 : Std_uLogic_Vector(7 downto 0) :=
    To_StdULogicVector(Std_Logic_Vector'(X"E4"));
-- VHDL'87 yields the following error: 
-- Illegal qualified expression.
-- Type error in bit string literal. Type Std_Logic_Vector is
-- not an array of bit.
-- VHDL'93 Works fine. 
end Confused;

```

**CASE 3**

**Figure 1.8.1 Error in of Std\_Logic\_1164 when Compiled with VHDL'87  
(element/confused.vhd)**

**A** The difference between the VHDL'87 and VHDL'93 languages is that the X"E4" is considered a bit string literal. In the 87 LRM, the X"E4" can only be used for *Bit\_Vector* types. In the 93 LRM, the X"E4" can be either *Bit\_Vector* or *Std\_Logic\_Vector*, as determined by context. See section 7.3.1 in both LRMs and note the differences in wording regarding *bit\_string\_literals*.<sup>[6]</sup>

First case. The function *To\_StdULogicVector(X"E4")* works in VHDL'87 because the argument can only be interpreted as a *Bit\_Vector*. In VHDL'93, the argument can be either *Bit\_Vector* or *Std\_Logic\_Vector* type. The compiler does not know which overload function to use.

Second case. This case works for both VHDL'87 and VHDL'93. The ambiguity is removed because of the casting to a type that works for both LRMs.

Third case. The *Std\_Logic\_Vector'(X"E4")* qualified expression is illegal in 87 because the operand is not the same type as the base type of the type mark. That is, *Bit* is not the same base type as *Std\_Logic*, and X"E4" is a *Bit\_Vector*. In VHDL'93, the operand type can be *Std\_Logic* as determined by the context.

### 1.7.2 Accessing Multiple Fields of Std\_Logic\_Vector

**Q** How can the fields of an object of type *Std\_Logic\_Vector* be accessed as enumeration literals (e.g., *ADD*, *SUB*, *PassA*, *PassB*, ...)? This approach would enhance readability. Type conversion functions can be used to achieve the desired result. What other techniques can be used to minimize the number of conversion functions?

### 1.7.3 Aliases and Constants

**A** An alias declares an alternate name for all, or part of an existing object, and enhances readability. The aliased name is a slice of the parent object. For example, figure 1.8.3-1 demonstrates the use of aliases for fields definition. Aliases are not supported in synthesis. An object cannot be aliased into a set of enumeration literals for use in statements like the *case* or *if* statement. One solution is the use of constant definitions as shown in figure 1.8.3-2. Note that the use of constants for the definition of

the “op-codes” is equivalent to binary or hex numbers, and do not represent enumeration literals.

```
library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.Std_Lock_Arith.all;

entity Cntlr is
    port(BusIn      : in  Std_Lock_Vector(31 downto 0);
          BusOut     : out Std_Lock_Vector(15 downto 0);
          Instr      : in  Std_Lock_Vector(2 downto 0));
    alias DataA_p  : Std_Lock_Vector(15 downto 0) is
                  BusIn(31 downto 16);
    alias DataB_p  : Std_Lock_Vector(15 downto 0) is
                  BusIn(15 downto 0);
end Cntlr;
```



**Figure 1.8.3-1 Uses of Aliases to Enhance Readability (\element\Cntlr\_e.vhd)**

```
architecture Cntlr_Const of Cntlr is
    constant PassHI_c   : Std_Lock_Vector(2 downto 0) := "000";
    constant PassLO_c   : Std_Lock_Vector(2 downto 0) := "001";
    constant AND_HiLo_c : Std_Lock_Vector(2 downto 0) := "010";
    constant OR_Hi_Lo_c : Std_Lock_Vector(2 downto 0) := "011";
    constant ADD_HiLo_c : Std_Lock_Vector(2 downto 0) := "100";
    constant SUB_HiLo_c : Std_Lock_Vector(2 downto 0) := "101";
    constant ZERO_HiLo_c: Std_Lock_Vector(2 downto 0) := "110";
    constant ONE_HiLo_c : Std_Lock_Vector(2 downto 0) := "111";

begin
    ArithLogic_Lbl: process(BusIn, Instr)
    begin
        case Instr is
            when PassHI_c    => BusOut <= DataA_p;
            when PassLO_c    => BusOut <= DataB_p;
            when AND_HiLo_c  => BusOut <= DataA_p and DataB_p;
            when OR_Hi_Lo_c  => BusOut <= DataA_p or DataB_p;
            when ADD_HiLo_c  =>
                BusOut <= Unsigned(DataA_p) +
                           Unsigned(DataB_p);           ← "+" and "-" are
                                                ← overloaded operator
                                                defined in
                                                IEEE.Std_Lock_Arith
            when SUB_HiLo_c  =>
                BusOut <= Unsigned(DataA_p) -
                           Unsigned(DataB_p);           ← "+" and "-" are
                                                ← overloaded operator
                                                defined in
                                                IEEE.Std_Lock_Arith
            when ZERO_HiLo_c => BusOut <= (others => '0');
            when ONE_HiLo_c  => BusOut <= (others => '1');
            when others       => BusOut <= (others => 'X');
        end case;
    end process ArithLogic_Lbl;
end Cntlr_Const;
```

**Figure 1.8.3-2 Uses of Aliases and Constants to enhance readability (\element\constant.vhd)**

The use of constants requires many definitions, and may be error prone. The compiler cannot check the definitions for accuracy and uniqueness.

### 1.7.4 Enumeration Type

The *Std\_Logic\_Vector* fields can be converted to the enumerated data type using one of the two methods shown in the next two subsections.

#### 1.7.4.1 Separate conversion functions.

This can be quite cumbersome when the number of conversions is excessive. An example of a synthesizable conversion function is shown in figure 1.8.4.1-1. A non-synthesizable version of this program that uses aliases is on disk in file (element\convert.vhd).

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_Arith.all;
entity Cntlr is
  port(BusIn      : in  Std_Logic_Vector(31 downto 0);
        BusOut     : out Std_Logic_Vector(15 downto 0);
        Instr      : in  Std_Logic_Vector(2  downto 0));
end Cntlr;

architecture Cntlr_Convert of Cntlr is  -- 
type Instr_Typ is (PassHi, PassLo, AND_Hilo, OR_Hi_Lo,
                    ADD_Hilo, SUB_Hilo, ZERO_Hilo, ONE_Hilo);
function ToInstr(Data : Std_Logic_Vector) return Instr_Typ is
  variable Data_v : Std_Logic_Vector(Data'length - 1 downto 0);
begin
  Data_v := Data;
  case Data_v(2 downto 0) is
    when "000" => return PassHi;
    when "001" => return PassLo;
    when "010" => return AND_Hilo;
    when "011" => return OR_Hi_Lo;
    when "100" => return ADD_Hilo;
    when "101" => return SUB_Hilo;
    when "110" => return ZERO_Hilo;
    when "111" => return ONE_Hilo;
    when others => return One_Hilo;
  end case;
end ToInstr;

begin
ArithLogic_Lbl: process(BusIn, Instr)
  variable Instr_v : Instr_Typ;
begin
  Instr_v := ToInstr(Instr);      -- function call
  case Instr_v is
    when PassHi      => BusOut <= BusIn(31 downto 16);
    when PassLo      => BusOut <= BusIn(15 downto 0);
    when AND_Hilo    => BusOut <= BusIn(31 downto 16) and
                           BusIn(15 downto 0);
    when OR_Hi_Lo   => BusOut <= BusIn(31 downto 16) or
                           BusIn(15 downto 0);
    when ADD_Hilo   =>
      BusOut <= Unsigned(BusIn(31 downto 16)) +
                  Unsigned(BusIn(15 downto 0));
    when SUB_Hilo   =>

```

```

        BusOut <= Unsigned(BusIn(31 downto 16)) -
                    Unsigned(BusIn(15 downto 0));
      when ZERO_HiLo  => BusOut <= (others => '0');
      when ONE_HiLo   => BusOut <= (others => '1');
      when others      => BusOut <= (others => 'X');
    end case;
  end process ArithLogic_Lbl;
end Cntlr_Convert;

```

**Figure 1.8.4.1-1 Conversion function to Enhance code Readability  
(element\convsynt.vhd)**

Note that Synopsys® synthesis tools allow for the use of the *ENUM\_ENCODING* attributes to encode the enumeration literals to values other than the standard ordering (i.e., 0 for 1st entry, 1 for the second, etc). If the use of such attributes is used, then a conversion function would be necessary to convert between a bit vector and an enumeration type. An example of *ENUM\_ENCODING* is shown in figure 1.8.4.1-2.

```

attribute ENUM_ENCODING : String; -- attribute definition
type Instr_Typ is (PassHi, PassLo, AND_HiLo, OR_Hi_Lo,
                    ADD_HiLo, SUB_HiLo, ZERO_HiLo, ONE_HiLo);
attribute ENUM_ENCODING of Instr_Typ: type is -- attribute
  "00000001 00001000 00000010 00000100" & -- declaration
  "10000000 01000000 00100000 00010000" ;

```

**Figure 1.8.4.1-2 Use of ENUM\_ENCODING Attribute**

#### 1.7.4.2 Use of Integer Subtypes

Integer subtypes can be used to identify ranges that can be used to access fields of bit vectors. For example:

```

subtype Source_Typ is Integer range 20 downto 18;
  if Instruction_Reg(Source_Typ) = "101" then ...

```

See sections 9.9.6 and 10.2.4 for more details.

#### 1.7.4.3 Use of 'val' attributes

This approach converts the bit vector into an integer, and uses this integer as an index to the value of the desired type. The  $T'val(X)$  is a function, with a result of base type  $T$ , that returns the value whose position number corresponds to  $X$ . This approach thus requires a set of enumerated type declarations and conversions from *Std\_Logic\_Vector* to *integer* (available in packages *IEEE.Std\_Logic\_Arith* and *IEEE.Numeric\_Std*). Figure 1.8.4.3 demonstrates this concept. The use of '*Val*' attribute is typically not synthesizable.

```

architecture Cntlr_Attr of Cntlr is
  type Instr_Typ is (PassHi, PassLo, AND_Hilo, OR_Hi_Lo,
                      ADD_Hilo, SUB_Hilo, ZERO_Hilo, ONE_Hilo);
begin
  ArithLogic_Lbl: process(BusIn, Instr)
    variable Instr_v : Instr_Typ;
    begin
      Instr_v := Instr_Typ'Val(CONV_INTEGER(UNSIGNED(Instr)));
      case Instr_v is
        when PassHi      => BusOut <= DataA_p;
        when PassLo      => BusOut <= DataB_p;
        when AND_Hilo    => BusOut <= DataA_p and DataB_p;
        when OR_Hi_Lo    => BusOut <= DataA_p or DataB_p;
        when ADD_Hilo    =>
          BusOut <= Unsigned(DataA_p) + Unsigned(DataB_p);
        when SUB_Hilo    =>
          BusOut <= Unsigned(DataA_p) - Unsigned(DataB_p);
        when ZERO_Hilo   => BusOut <= (others => '0');
        when ONE_Hilo    => BusOut <= (others => '1');
        when others      => BusOut <= (others => 'X');
      end case;
    end process ArithLogic_Lbl;
end Cntlr_Attr;

```

*'Val* requires an integer index.  
 IEEE.Std\_Logic\_Arith provides a conversion  
 function for Unsigned type. Type conversion from  
 Std\_Logic\_Vector to Unsigned is required.



**Figure 1.8.4.3 Use of '*Val* to Enhance Readability (element\attribute.vhd)**

### 1.7.5 Testbench for Demonstration Models

Figure 1.8.5-1 demonstrates a simple testbench to verify the operations of these various modeling alternatives. Figure 1.8.5-2 demonstrates the use of configuration declarations for the various architectures. Figure 1.8.5-3 demonstrates the compilation script using Model Technology V-System compiler<sup>6</sup>.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_Arith.all;
entity CNTLRTB is
begin
end CNTLRTB;

architecture CNTLRTB_A of CNTLRTB is
component Cntlr
  port(
    BusIn      : in  Std_Logic_Vector(31 downto 0);
    BusOut     : out Std_Logic_Vector(15 downto 0);
    Instr      : in  Std_Logic_Vector(2  downto 0));
  end component;
  signal BusIn : Std_Logic_Vector(31 downto 0);
  signal BusOut : Std_Logic_Vector(15 downto 0);
  signal Instr : Std_Logic_Vector(2  downto 0) := "111";

```

```

begin
  U1_Cntlr: Cntlr
    port map (
      BusIn      => BusIn,
      BusOut     => BusOut,
      Instr      => Instr);

  DriveData_Lbl: process

  begin
    BusIn <= "01110010101100011111000001011101";
    -- from IEEE. Std_Logic_Arith
    -- function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    Instr <= Unsigned(Instr) + 1;
    wait for 50 ns;
  end process DriveData_Lbl;
end CNTLRTB_A;

```

**Figure 1.8.5-1 Testbench for Controller Model (element\cntlrb.vhd)**

```

configuration CNTLR_Constant_Cfg of CNTLRTB is
  for CNTLRTB_A
    for U1_Cntlr : Cntlr use
      entity Work.Cntlr(Cntlr_Const);
    end for;
  end for;
end CNTLR_Constant_Cfg;

configuration CNTLR_Attr_Cfg of CNTLRTB is
  for CNTLRTB_A
    for U1_Cntlr : Cntlr use
      entity Work.Cntlr(Cntlr_Attr);
    end for;
  end for;
end CNTLR_Attr_Cfg;

configuration CNTLR_Convert_Cfg of CNTLRTB is
  for CNTLRTB_A
    for U1_Cntlr : Cntlr use
      entity Work.Cntlr(Cntlr_Convert);
    end for;
  end for;
end CNTLR_Convert_Cfg;

```

**Figure 1.8.5-2 Configuration Declarations (element\cntlr\_c.vhd)**

```

vcom  cntlr_e.vhd
vcom  constant.vhd
vcom  convert.vhd
vcom  attribte.vhd
vcom  cntlrb.vhd
vcom  cntlr_c.vhd

```

**Figure 1.8.5-3 Compilation file using Model Technology V-System Compiler**

## 1.8 RANGE CONSTRAINT IN TYPE DEFINITION

**Q** | LRM reference 3.2.1, line 124 that states: [1] Each bound of a range constraint that is used in an integer type definition must be a locally static expression of some integer type. On line 119 of the same section, the LRM states: [1] An integer type definition defines both a type and a subtype of that type. Given those definitions, how come line 6 of the following code compiles properly since the range A\_g is globally static and a subtype is defined? Note that Line 7 fails as expected because of the LRM requirements.

```
entity T is
  generic(A_g : Integer);
end T;

architecture T_a of T is
  subtype X_Typ is Integer range 0 to A_g; -- line 6
  type MyInt_Type is range 0 to A_g; -- line 7
  -- # ERROR: H:/BEN/TEST3.VHD(7):
  --     Range constraint in type definition must be locally static.
begin
end T_a;
```

Figure 1.9 Subtype Definition (element\static.vhd)

**A** | Even though line 6 defines a subtype, that line is defining a base type (type *integer* in this case). The range of the type must be locally static, and it is in this case because of the definition of type *integer*. The LRM makes this limitation on types so that the compiler can figure out how much storage to allocate to hold the type. For subtypes, since the range can never be larger than that of the base type, it already knows the allocation size and it does not need to be locally static. Thus, in the example, the subtype definition (in line 6) is acceptable, whereas the type definition (in line 7) is in error.

## 1.9 SHARED VARIABLES

**Q** | In a multi-threaded application, how can processes that assign values to shared variables be assured to have exclusive use of those variables?

**A** | Shared variables are immediately updated because they represent variables rather than signals. In a multi-threaded application, multiple processes can access shared variables. Since the simulation order of processes is not specified in the language, one cannot rely on the processing order to guarantee exclusivity in the use of a shared variable. A semaphore scheme is required. One such technique is the use of a signal of a resolved integer type to be used as *bus requests* by the requesting processes, and as *bus grants* by the arbiter. In this case, the resolved integer resolution function provides priority to the process driving the highest value onto a signal of that type. Every process capable of participating in the arbitration is assigned an exclusive integer number (Process ID). This technique is independent on the number of processes. The protocol is as follows:

- Any process that does not need the shared resource in the current simulation time assigns a value of *Integer'low* onto the resolved integer signal (that is the arbitration signal). This would be equivalent to a deallocation.
- If a process wants to allocate the resource, then it must first wait until the value of *Integer'low* is on that arbitration signal. This is to insure that no process is using that resource.
- Once the arbitration signal has attained this idle value, any process requesting access to the shared resource assigns its exclusive integer number onto that resolved arbitration signal. The resolution function will automatically resolve the final value to the highest driven integer value.
- Any process that made the request access must then wait until the value on the arbitration signal matches its assigned process ID. Once a match occurs, the process may use, and may hold onto the shared resource for as long as required.
- When the resource is no longer required, that process must assign the value of *Integer'low* onto the arbitration signal to provide for the deallocation. This enables other processes that made the request to get access to the shared resource.

Figure 1.10 provides a complete example of two processes accessing a shared variable. In this example, a package is used to define the integer resolution function. The shared variable is also declared in this package, and thus may be used by other components.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

package Shared_Pkg is
    type Complex_Typ is record
        R      : Real;
        I      : Integer;
        V      : Std_Logic_Vector(7 downto 0);
    end record;
    shared variable Shared_v : Complex_Typ :=
        (R => 10.0,
         I => 7,
         V => "10001111"); -- shared variable
    type Int_Array is array(Integer RANGE <>) of Integer;
    -- High wins over Low
    function rInteger(DRIVERS : Int_Array) return Integer;
    subtype rInteger_Typ is rInteger Integer;
    signal rInt_s : rInteger_Typ;
end Shared_Pkg;

package body Shared_Pkg is
    function rInteger(DRIVERS : Int_Array) return Integer is
        variable Highest : Integer := Integer'low;
    begin
        LoopThruAllDrivers: for I in DRIVERS'range loop
            if DRIVERS(I) > Highest then
                Highest := DRIVERS(I); -- take the highest
            end if;
        end loop LoopThruAllDrivers;
        return Highest;
    end rInteger;
end Shared_Pkg;
-----
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;

library Work;
use Work.Shared_Pkg.all;
use Work.LfsrStd_Pkg.all; -- See section 5.4

entity TShared is
end TShared;

architecture TShared_a of TShared is
    signal Clk      : Std_Logic := '1';
    signal Rand_s  : Std_Logic_Vector(5 downto 0) := "101001";
    signal S1       : Complex_Typ; -- for display of assigned values
    signal S2       : Complex_Typ; -- for display of assigned values
    signal P1Want   : Boolean;    -- for display, Process 1 wants the bus
    signal P2Want   : Boolean;    -- for display, Process 2 wants the bus

```

Type for shared variable. Simple record used here for demonstration. Type could be more complex and could represent a large memory array.

Shared variable declaration.

Global signal of resolved integer type used for the resource requests and for arbitration.

Signal used for random number

```

begin -- TInteger_a
T1_Lbl : process
  constant T1_c : Integer := 10; -- Process ID number
begin -- process T2_Lbl
  -- wait for a pseudo random number of cycles
  for I in 1 to Conv_Integer(Unsigned(Rand_s)) loop
    wait until Clk'event and Clk = '1'; -- for synchronous action
  end loop;

  -- Wait until no process is allocating the shared resource
  P1Want <= True;
  Wait4Idle_Lbl : while Work.Shared_Pkg.rInt_s /= Integer'low loop
    wait until Clk'event and Clk = '1'; -- for synchronous action
  end loop Wait4Idle_Lbl;

  -- allocate the resource
  Work.Shared_Pkg.rInt_s <= T1_c;
  -- check for resource grant
  wait until Work.Shared_Pkg.rInt_s = T1_c;

  -- Resource is allocated
  P1Want <= False;
  Work.Shared_Pkg.Shared_v.R := 1.0;
  Work.Shared_Pkg.Shared_v.I := Work.Shared_Pkg.Shared_v.I + 1;
  Work.Shared_Pkg.Shared_v.V := not Work.Shared_Pkg.Shared_v.V;
  S1 <= Work.Shared_Pkg.Shared_v;

  -- Keep hold of shared resource for a pseudo random number of cycles
  for I in 1 to Conv_Integer(Unsigned(Rand_s)) loop
    wait until Clk'event and Clk = '1'; -- for synchronous action
  end loop;

  assert Work.Shared_Pkg.Shared_v.I = S1.I -- check for no change in value
    report "P1: Shared resource modified by another resource"
    severity Warning;
  -- deallocate shared resource
  Work.Shared_Pkg.rInt_s <= Integer'low; -- end of request;
end process T1_Lbl;

T2_Lbl : process
  constant T2_c : Integer := 5; -- Process ID number
begin -- process T2_Lbl
  -- wait for a pseudo random number of cycles
  for I in 1 to Conv_Integer(Unsigned(Rand_s)) loop
    wait until Clk'event and Clk = '1'; -- for synchronous action
  end loop;

```

**Allocation protocol:** Wait until no process is using the resource, then make a request and wait until access to the resource is granted

Modify resource at will, and hold resource until done.

Deallocate the shared resource

```

-- Wait until no process is allocating the shared resource
P2Want <= True;
Wait4Idle_Lbl : while Work.Shared_Pkg.rInt_s /= Integer'low loop
    wait until Clk'event and Clk = '1'; -- for synchronous action
end loop Wait4Idle_Lbl;

-- allocate the resource
Work.Shared_Pkg.rInt_s <= T2_c;                                ↗
-- check for resource grant
wait until Work.Shared_Pkg.rInt_s = T2_c;                         ↘

Allocation protocol: Wait until no process is
using the resource, then make a request and
wait until access to the resource is granted

-- resource is allocated
P2Want <= False;
Work.Shared_Pkg.Shared_v.R := 2.0;
Work.Shared_Pkg.Shared_v.I := Work.Shared_Pkg.Shared_v.I + 10;
Work.Shared_Pkg.Shared_v.V := not Work.Shared_Pkg.Shared_v.V;
S2 <= Work.Shared_Pkg.Shared_v;

-- Keep hold of shared resource for a pseudo random number of cycles
for I in 1 to Conv_Integer(Unsigned(Rand_s)) loop
    wait until Clk'event and Clk = '1'; -- for synchronous action
end loop;

assert Work.Shared_Pkg.Shared_v.I = S2.I -- check for no change in value
report "P2: Shared resource modified by another resource"
severity Warning;

-- deallocate shared resource
Work.Shared_Pkg.rInt_s <= Integer'low; -- end of request;
end process T2_Lbl;

Random_Lbl : process
begin -- process Random_Lbl
    wait until Clk'event and Clk = '1';
    Rand_s <= LFSR(Rand_s); -- change seed
end process Random_Lbl;

Clk <= not Clk after 25 ns;
end TShared_a;

```

**Application of LFSR to generate a random number within the bit range of Rand\_s.**

**Figure 1.10 Example of Two Processes Accessing a Shared Variable  
(element\shared.vhd)**

## **2. ARRAYS**

---

---

Arrays are important data structures in VHDL because they represent busses, registers, and memories. The language provides several rules regarding the manipulation of arrays. This section addresses many of those issues, including array operations, array initialization, use of constrained and unconstrained arrays, and mapping of arrays of different sizes. The application of arrays in synthesis is also addressed.

## 2.1 ARRAY STRUCTURE REPRESENTATIONS

**Q** | What structures do arrays represent? How can they be used? What is their significance in synthesis?

### 2.1.1 One Dimensional Arrays

**A** | One dimensional arrays define vectors. In synthesis, one dimensional arrays represent either the output of combinational logic or a register or a latch. See section 7.3 (latch/register/combinational logic) for a discussion of how one-dimensional arrays are used.

### 2.1.2 Multi-Dimensional Arrays

Multi-dimensional arrays are arrays with more than one index. For example, an object of type "*A2D5x5\_Typ is array(5 downto 0, 4 downto 0) of Std\_Logic;*" represents a two-dimensional array (*X*, *Y*), where one index has six entries, and the other index has five entries. An object of this type is like a checkerboard with a total of thirty individual elements, where each element represents one bit (of type *Std\_Logic*). Two dimensional arrays are addressed as ROW first, and then COLUMN next. (An easy method to remember the Row/Column addressing sequence is to think of the familiar *RC* time constant, or *RC® Cola*.)

This structure is often used to describe several conditions that define a single parameter. One such example, used in the package *Std\_Logic\_1164*, is the resolution constant as shown in figure 2.1.2-1. The constant is used as a lookup table.

```
type Stdlogic_Table is array(Std_uLogic, Std_uLogic) of
    Std_uLogic;
    |
    |----- Row -----|----- Column -----|
constant Resolution_Table : Stdlogic_Table := (
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
-----| U   | X   | 0   | 1   | Z   | W   | L   | H   | -   | I   | I   |
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
-----| ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), -- | U |
-----| ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- | X |
-----| ('U', 'X', '0', 'X', '0', '0', '0', '0', '0', 'X'), -- | 0 |
-----| ('U', 'X', 'X', '1', '1', '1', '1', '1', '1', 'X'), -- | 1 |
-----| ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'W', 'Z'), -- | Z |
-----| ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W', 'W'), -- | W |
-----| ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'W', 'L'), -- | L |
-----| ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'W', 'H'), -- | H |
-----| ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X')  -- | - |
);
```

**Figure 2.1.2-1 Two-Dimensional Array Used in *Std\_Logic* Resolution Function**

Another example of a two-dimensional lookup table is the time delay for each pin of a component (the Row), and the environmental condition (the Column). Figure 2.1.2-2

demonstrates the use of two-dimensional arrays to specify the time parameters of a design.

```

entity TimeSpec_Nty is
  port (Address : out      Integer;
        Data      : inout    Integer;
        Control   : out      Integer);
end TimeSpec_Nty;

architecture TimeSpec_a of TimeSpec_Nty is
  type TimeParam_Typ is
    (tpd_Address,     tpd_Data,      tpd_Control,
     tsetUp_Address,  tsetUp_Data,   tsetUp_Control,
     tHold_Address,   tHold_Data,   tHold_Control);
  type Cond_Typ is (tMin, tTyp, tMax);
  type tSpec_Typ is array(TimeParam_Typ, Cond_Typ) of time;

  constant Spec_c : tSpec_Typ :=
    (tpd_Address    => (tMin => 20 ns, tTyp => 25 ns, tMax => 30 ns),
     tpd_Data       => (tMin => 21 ns, tTyp => 26 ns, tMax => 30 ns),
     tpd_Control    => (tMin => 22 ns, tTyp => 27 ns, tMax => 30 ns),

     tsetUp_Address => (tMin => 23 ns, tTyp => 25 ns, tMax => 30 ns),
     tsetUp_Data    => (tMin => 24 ns, tTyp => 25 ns, tMax => 30 ns),
     tsetUp_Control => (tMin => 25 ns, tTyp => 25 ns, tMax => 30 ns),

     tHold_Address  => (tMin => 1 ns,   tTyp => 2 ns,   tMax => 3 ns),
     tHold_Data     => (tMin => 1 ns,   tTyp => 2 ns,   tMax => 3 ns),
     tHold_Control  => (tMin => 1 ns,   tTyp => 2 ns,   tMax => 3 ns));

begin -- TimeSpec_a
  -- Process: Test_Lbl
  -- Purpose: Demonstrate use of timing parameters
  Test_Lbl : process
  begin -- process Test_Lbl
    Address <= 100 after Spec_c(tpd_Address, tTyp);
    Control  <= 2  after Spec_c(tpd_Control, tTyp);
    Data     <= 25 after Spec_c(tpd_Data, tTyp);
    wait;
  end process Test_Lbl;
end TimeSpec_a;

```

**Figure 2.1.2-2 Time Parameters with Two-dimensional Arrays  
(arrays\tmspec.vhd )**

Generally, multi-dimensional arrays are not allowed in synthesis for the definition of signals or variables. They are allowed as constants or table lookups. One way around this restriction is to declare one-dimensional arrays of constrained one dimensional array. For example:

```

subtype Data_Typ is Std_Logic_Vector(31 downto 0);
type Mem2_Typ is array(0 to 511) of Data_Typ;

subtype MemSize_Typ is Integer range 0 to 511; -- optional
type Mem_Typ is array(MemSize_Typ) of Data_Typ; -- defintion

```

This approach is easier to use and is more representative of actual hardware. Section 7.8 demonstrates the declaration of a synthesizable register file using this concept.

## 2.2 ARRAYS -- LEGAL OPERATIONS

**Q**

What are the implicit array operations? What are the overloaded operators on arrays? Why do we need them?

**A**

When an array type is declared, several implicit functions are automatically declared for operations on this type. These functions are shown in table 2.2.

**Table 2.2 Implicit Operations on Arrays<sup>7</sup>**

OPERATION	RESTRICTIONS ON LEFT AND RIGHT OPERANDS
Assignments :=, <=	Must be the same size and type.
Relational <, >, <=, >=	<p>Must be one-dimensional discrete arrays of the same type.            Per LRM 7.2.2, [1] for discrete array types, the relation &lt; (less than) is defined such that the left operand is less than the right operand if and only if:</p> <ol style="list-style-type: none"> <li>1. The left operand is a null array and the right operand is a non-null array; otherwise,</li> <li>2. Both operands are non-null arrays, and one of the following conditions is satisfied:               <ol style="list-style-type: none"> <li>a. The leftmost element of the left operand is less than that of the right; or</li> <li>b. The leftmost element of the left operand is equal to that of the right, and the tail of the left operand is less than of the right (the tail consists of the remaining elements to the right of the leftmost element and can be null).</li> </ol> </li> </ol> <p>Another way of stating condition 2 is as follows:            Each array component is compared, in turn, from left to right, until a difference is found, or until all the components of one or both arrays have been exhausted. If a difference in a corresponding component is detected, then the array possessing the component with the smaller value is considered to be "less than" the other array, regardless of what component may follow. Thus, "DO" &lt; "IF", and (2,5) &lt; (9, 3). Consider the case when all the components of one array have been exhausted and no difference in a corresponding component has been detected. If additional components remain for the second array, then the first array is considered to be "Less than" the second array. Thus, "DO" &lt; "DONE", and (2,5) &lt; (2,5,2,3)</p>
Equality =	Same type, number of components, and all corresponding components are equal. "DONE" = "DONE", (3,1,3) = (3,1,3)
Inequality /=	Each array component is compared, in turn, from left to right, until a difference is found, or until all the components of one or both arrays have been exhausted. If a difference in a corresponding component is detected, then the arrays are not equal. If all the components of one array have been exhausted and no difference in a corresponding component has been detected, and if additional components remain for the second array, then the arrays are unequal. "DONE" /= "DON", (3,1,3) /= (3,1,1)
Concatenation &	Left and right side must be one-dimensional arrays of the same type. A := "VHDL" & " IS" & " GREAT"; – A is of a string type
Slicing	Array slicing must be one-dimensional array. A(7 downto 0) := B(31 downto 24);

Logical operator	Predefined operators (and not implicit) for objects of type bit, Boolean, and one dimensional arrays of Bit and Boolean elements include: <b>not, and, or, nand, nor, xor, xnor</b> (VHDL'93).
Shift operator	VHDL'93 Bit_Vector shift operators include: (These operators are not implicit operators) <b>sll, srl, sla, sra, rol, ror</b>
Array attributes	<b>A'left, A'right, A'high, A'low, A'range, A'reverse_range, A'length, (A'ascending, - '93)</b> (where A represents the array object)
Type conversion.	An array can be converted to another type of array only when each array has the same shape and size, the same component type, and the same or convertible corresponding index types. Basically, the arrays must be "closely related." <pre>signal S : Unsigned(7 downto 0); signal B : Std_Logic_Vector(7 downto 0) := "1000_0000"; ... S &lt;= Unsigned(B);</pre>

Figure 2.2 demonstrates examples of implicit operations on arrays.

```

library IEEE;
use IEEE.Std_Lock_1164.all;
entity Array_Nty is
end Array_Nty;

architecture Array_a of Array_Nty is
  subtype A5_Typ    is Std_Lock_Vector(4 downto 0);
  subtype A7_Typ    is Std_Lock_Vector(8 downto 2);
  type Mem5x5_Typ is array(4 downto 0) of A5_Typ;

  signal A5_s      : A5_Typ      := (others => '0');
  signal A7_s      : A7_Typ      := (others => '0');
  signal MemA_s   : Mem5x5_Typ := (others => (others => '0'));

begin
  Test_Lbl: process
    variable A5_v      : A5_Typ      := (others => '1');
    variable A7_v      : A7_Typ      := (others => '1');
    variable MemA_v   : Mem5x5_Typ := (others => (others => '1'));
    variable Int_v     : Integer;
  begin
    -- Assignment :=, <=
    A5_v := "10101";
    A5_s <= A5_v;

    MemA_v(0) := "10101";
    A7_s(8 downto 4) <= MemA_v(0);

    --equality
    assert not (MemA_v(0) = A5_v)
      report "MemA_v(0) = A5_v"          -- true
      severity note;

    MemA_s <= MemA_v; -- assignment of whole aggregate
    wait for 10 ns;
    assert not (A5_s = A5_v)
      report "A5_s = A5_v"            -- true
      severity note;
  end process;
end;

```

```

assert not (MemA_s = MemA_v)
report "MemA_s = MemA5_v" -- true
severity note;

-- Relational Operators <, <=, >, >=
A5_s <= "10110"; -- note that A5_v = "10101"
wait for 10 ns;

assert not (A5_s > A5_v) -- "10110" > "10101"
report "A5_s > A5_v"
severity note;

A7_v := "1010100";
-- A5_v < A7_v because "10101" < "1010100" -- LRM 7.2.2
assert not (A5_v < A7_v)
report "A5_v < A7_v"
severity note;

-- Inequality
assert not (A5_v /= A7_v)
report "A5_v /= A7_v"
severity note;

-- null array < non null array
assert not (A7_v(2 to 8) < A5_v)
report "A7_v < A5_v"
severity note;

-- Concatenation
A7_v := A5_v(4) & '1' & A5_v(2 downto 0) & "00";

-- Array attributes (A represents the array object)
-- A'left, A'right, A'high, A'low, A'range,
-- A'reverse_range, A'length
-- A'ascending, -- '93
Range_Lbl: for I in A5_v'range loop
    A5_v(A5_v'high - I) := A5_s(I); -- bit reverse
end loop Range_Lbl;
end process Test_Lbl;
end Array_a;

```

Null slice in VHDL'87,  
ERROR in VHDL'93 (See LRM 6.5  
and section 2.3)

**Figure 2.2 Implicit Operations on Arrays, (arrays\array1.vhd)**

### 2.2.1 Overloaded Operators on Arrays

Overloaded operators allow for user defined interpretations of operations on objects. For example, the package *Std\_Logic\_1164* overloads the logical operators (**and**, **nand**, **or**, **nor**, **xnor**, **not**) for objects of type *Std\_uLogic*, and *Std\_uLogic\_Vector*, and *Std\_Logic\_Vector*. Other examples of overloaded operators are the operations on objects of *Signed* and *Unsigned* data types (e.g., "+", "-", "=", "/="). For objects of these types the implicit equality, inequality and relational operator do not apply. For example, a *Signed* object of value "1111010" is equal to a *Signed* object of value "1010"; a *Signed* object of value "1010" (-6) is less than a *Signed* object of value "0101" (+5). The implicit relational operators do not apply to those types. Type *Signed* and *Unsigned* are defined in *Std\_Logic\_Arith* and *Numeric\_Std* packages and have the following overloaded functions "+", "-", "\*", *ABS*, "<", "<=", ">", ">=", "=", "/=". A sample overloaded function

“<” extracted from the *Std\_Loic\_Arith* package is shown in figure 2.2.1-1. Note that the equality operators for *Signed* and *Unsigned* types have an implicit definition (as the result of the enumeration type declaration) and an explicit definition in the package. Many vendors have adopted a compilation option (- *explicit*) to direct the compiler to resolve this ambiguous function overloading in favor of the explicit function definition.

```

function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN is
  constant Length: INTEGER := Max(L'length, R'length);
begin
  return Unsigned_Is_Less(CONV_UNSIGNED(L, length),
    CONV_UNSIGNED(R, length));
end;

function "<"(L: SIGNED; R: SIGNED) return BOOLEAN is
  constant length: INTEGER := Max(L'length, R'length);
begin
  return Is_Less(CONV_SIGNED(L, length),
    CONV_SIGNED(R, length));
end;

```

Figure 2.2.1-1 Overloaded Function “<”

Figure 2.2.1-2 provides application examples of the overloaded “+” operator on *signed* and *unsigned* objects. The model makes use of the *Image\_Pkg*, described in section 5.1, to provide outputs to the screen. Figure 2.2.1-3 represents the results of a simulation run using Model Technology’s *VSystem*.

```

library IEEE;
use IEEE.Std_Loic_Arith.all;
use IEEE.Std_Loic_1164.all;
library Work;
use Work.Image_Pkg.all; ← See section 5.1 for description of package
library Std;
use Std.TextIO.all;
entity SgnUsgn is
end SgnUsgn;

architecture SgnUsgn_a of SgnUsgn is
begin X_Lbl: process
  variable D1_v : Std_Loic_Vector(7 downto 0) := "00000010"; -- 2
  variable D2_v : Std_Loic_Vector(6 downto 0) := "1111000"; -- +120
  variable N1_v : Signed(7 downto 0) := "00000010"; -- 2
  variable N2_v : Signed(6 downto 0) := "1111000"; -- -8
  variable U1_v : Unsigned(7 downto 0) := "00000010"; -- 2
  variable U2_v : Unsigned(6 downto 0) := "1111000"; -- +120
  variable Line_v : Std.TextIO.Line;
begin

```

```

if D1_v > D2_v then
  assert False
    report "D1_v > D2_v" & "  D1_v = " & Image(D1_v) &
          "  D2_v = " & Image(D2_v)
  severity Note;

else
  assert False
    report "D1_v < D2_v" & "  D1_v = " & Image(D1_v) &
          "  D2_v = " & Image(D2_v)
  severity Note;
end if;

if N1_v > N2_v then
  assert False
    report "N1_v > N2_v" & "  N1_v = " & Image(N1_v) &
          "  N2_v = " & Image(N2_v)
  severity Note;
else
  assert False
    report "N1_v < N2_v" & "  N1_v = " & Image(N1_v) &
          "  N2_v = " & Image(N2_v)
  severity Note;
end if;

if U1_v > U2_v then
  assert False
    report "U1_v > U2_v" & "  U1_v = " & Image(U1_v) &
          "  U2_v = " & Image(U2_v)
  severity Note;
else
  assert False
    report "U1_v < U2_v" & "  U1_v = " & Image(U1_v) &
          "  U2_v = " & Image(U2_v)
  severity Note;
end if;

N1_v := Signed(D1_v); -- Type conversion
N1_v := N1_v + N2_v;  -- 2 + (-8) == "11111010"
Std.TextIO.Write(Line_v, String'("N1_v ="));
Std.TextIO.Write(Line_v, Image(N1_v));
Std.TextIO.WriteLine(Output, Line_v);

U1_v := U1_v + U2_v; -- +2 + 120 -- "01111010"
Std.TextIO.Write(Line_v, String'("U1_v ="));
Std.TextIO.Write(Line_v, Image(U1_v));
Std.TextIO.WriteLine(Output, Line_v);

D1_v := Std_Logic_Vector(U1_v); -- type conversion
D2_v := Std_Logic_Vector(N2_v); -- type conversion
wait;
end process X_Lbl;
end SgnUsgn_a;

```

**Figure 2.2.1-2 Application of the Overloaded “+” Operator on Signed and Unsigned Objects (arrays\signed.vhd)**

```

vsim
# Loading C:\VHDL\std.standard
# Loading C:\VHDL\ieee.std_logic_1164(body)
# Loading C:\VHDL\ieee.std_logic_arith(body)
# Loading C:\VHDL\std.textio(body)
# Loading C:\VHDL\ieee.std_logic_textio(body)
# Loading work.image_pkg(body)
# Loading work.sgnusgn(sgnusgn_a)
run 100
# ** Note: D1_v < D2_v  D1_v = 00000010  D2_v = 1111000
#   Time: 0 ns  Iteration: 0  Instance:/
# ** Note: N1_v > N2_v  N1_v = 00000010  N2_v = 1111000
#   Time: 0 ns  Iteration: 0  Instance:/
# ** Note: U1_v < U2_v  U1_v = 00000010  U2_v = 1111000
#   Time: 0 ns  Iteration: 0  Instance:/
# # N1_v =11111010
# # U1_v =01111010

```

**Figure 2.2.1-3 Simulation Run of *SgnUsgn\_a* Architecture**

A question that often arises is how can the “=” operator be overloaded for *Std\_uLogic* and *Std\_uLogic\_Vector* types? LRM 7.2.2 states: [1] Relational operator 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*. The equality and inequality operators (= and /=) are defined for all types other than file types. This means that once a type (except file type) is declared, the relational operators for that type are IMPLICITLY defined. All enumeration data types in VHDL get an implicit definition for the “=” operator. So while there is no explicit “=” operator, there is an implicit one. However, it is possible to overload any of the operators. For example, in package *Std.Logic\_Arith* distributed by Synopsys®, the “=” operator is overloaded for type *Unsigned* as shown in figure 2.2.1-4.

It is important to note that the VHDL local scoping rules also apply to the overloaded functions. Therefore, it is possible to define a local replacement to an operator by using the scope rules. For example, declaring an operator in an architecture will overload one declared externally in a package. Similarly, an operator declared in a process will overload the one declared in the architecture.

```

function "="(L: Unsigned; R: Unsigned) return Boolean is
  constant Length: Integer := Max(L'length, R'length);
begin
  return BitWise_Eql(Std_uLogic_Vector(CONV_UNSIGNED(L, Length)),
                      Std_uLogic_Vector(CONV_UNSIGNED(R, Length)));
end;

function BitWise_Eql(L: Std_uLogic_Vector; R: Std_uLogic_Vector)
                     return BOOLEAN is
begin
  for I in L'range loop
    if L(I) /= R(I) then
      return False;
    end if;
  end loop;
  return True;
end;

```

This function DOES make use of  
the implicit "/=" operator

**Figure 2.2.1-4 Overloaded Operator “=” for Type *Unsigned* in Package *Std\_Logic\_Arith***

LRM 2.3 states: [1] 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. For the operator subprograms, the question then arises: which subprograms are used in an operation, the implicit operators (implicitly defined by the type declarations), or the explicit operators (explicitly defined in functions)?

The key to the answer to this question is in the rules about homographs. [1] A homograph is 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. 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. When 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.

The way to usefully overload an implicitly declared operator is to declare a homograph of that operator in the same declarative region (e.g., the declarative region containing the type declaration that results in the implicit operator declaration). In this case, the user-defined homograph will completely hide the implicitly declared operator. It will be as if the implicitly declared operator ceased to exist.

It is also possible to declare a homograph of an implicitly declared operator in a different region. For instance, if an operator is implicitly declared in package *P*, then it is possible

to declare a homograph in package *Q*. However, it is necessary to call the function explicitly if another version of the implicit (e.g. “=”) operator is declared in a package other than the one containing the enumeration declaration, and both operators become visible through *use* clauses. For example,

```
lib.P."="(arg1, arg);
lib.Q."="(arg1, arg);
```

Certain VHDL analyzers take the approach that an explicitly declared declaration always overrides an implicitly declared homograph, but the LRM allows this hiding only if the two are declared in the same declarative region. Most simulators, including Model Technology’s VSystem®, provide the option to allow explicit operators to hide the implicit one. If selected, this option directs the compiler to resolve ambiguous function overloading in favor of the explicit function definition.

## 2.3 ARRAY SLICES AND RANGES

**Q**

Given the following declaration and signal assignments, what are the values of *X*(0), *X*(3), *Y*(0), *Y*(3) ? Is the range ignored in determining the index in a loop?

```
Signal X : Std_Logic_Vector(0 to 3);      -- CASE 1
signal Y : Std_Logic_Vector(3 downto 0);   -- CASE 2
X <= "1010";
Y <= "0101";
```

**A**

Assignment in VHDL preserves the left-to-right ordering regardless of the direction. In the first case, the string "1010" is assigned to the elements of *X*

from left to right. This means *X*(0) = '1', *X*(1) = '0', *X*(2) = '1', *X*(3) = '0'. Similarly, in the second case, *Y* is also assigned from left to right, so *Y*(3) = '0' and *Y*(0) = '1'. The assignment *X* <= *Y* is equivalent to:

```
X(0) <= Y(3); -- the left elements
X(1) <= Y(2);
X(2) <= Y(1);
X(3) <= Y(0); -- the right elements
```

The range is NOT ignored in determining the index of a loop. For example, if the index of the elements of *X* were from 0 to 3 (e.g. *for I in 0 to 3 loop*, or *for I in X’range loop*), then the elements of the arrays would be accessed from left to right. If the same loop were used for *Y*, then the elements of the array would be accessed from right to left instead. The *reverse\_range* attribute would cause the arrays to be accessed in the reverse direction (*3 downto 0* for *X*, and *0 to 3* for *Y*).

Note that slices of an array allow for the extraction of certain elements of the array. for example, for the above definition of *X* and *Y*, the following statements are true:

<i>X</i> (1 to 2) is "01"	<i>X</i> (1 to 1) is "0" – a one element array,
<i>Y</i> (2 downto 1) is "10"	<i>Y</i> (2 downto 2) is "1" – a one element array,

Null slices are slices with no values, but do not necessarily represent an error. There is a

difference between VHDL'87 and VHDL'93 in the definition of a *null slice*. Specifically, LRM 6.5 states the following:

### VHDL'87

[1]The slice is a *null slice* if the discrete range is a null range, or if the direction of the discrete range is not the same as that of the object denoted by the prefix of the slice name.

*Constant DATA : Bit\_Vector(31 downto 0);*

*DATA(24 to 25) -- a null slice* ☺--direction  
*DATA(24 downto 25) – a null slice,*

-- null range ☺

### VHDL'93

[1]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.

*Constant DATA : Bit\_Vector(31 downto 0);*

*DATA(24 to 25) -- an error* ♡--direction  
*DATA(24 downto 25) – a null slice*

-- null range ☺

Other examples of slices include:

- signal X : Std\_Logic\_Vector(0 to 3); – CASE 1

- signal Y : Std\_Logic\_Vector(3 downto 0); – CASE 2

X(2 downto 1) VHDL'87 null slice because direction of the discrete range is not the same as that of the object denoted by the prefix of the slice name (i.e. Slice direction is downward (**downto**) while slice name (X) is upward (**to**)).

It is an error in ♡ VHDL'93

Y(1 to 2) VHDL'87 null slice, ♡VHDL'93 (same as above)

X(1 downto 1) VHDL'87 null slice, ♡VHDL'93 (same as above)

Y(2 to 2) VHDL'87 null slice, ♡VHDL'93 (same as above)

X(2 to 1) Null slice because the discrete range is a null range 2 (is greater than 1)

Y(1 downto 2) Null slice because the discrete range is a null range (1 is less than 2)

Null slices are generally NOT synthesizable.

In non-synthesizable models, null arrays are useful when the describing arrays that may dynamically be null. An example of such a condition is demonstrated in the model of barrel shifter shown in section 7.11.

## 2.4 ARRAY INITIALIZATION

**Q**

The code in figure 2.4-1 yields compilation errors in the initialization of signals. Why does the compiler yield these errors?

```
architecture test of test is
    signal Clock1 : Bit_Vector(1 to 1) := (1 => '1');
    signal Clock2 : Bit_Vector(1 to 2) := (1 | 2 => '1');
    signal Clock3 : Bit_Vector(1 to 1) := (others => '1');
    signal Clock4 : Bit_Vector(1 to 2) := (others => '1');
    signal Clock5 : Bit_Vector(1 to 1) := (Clock5'range => '1'); -- line 9 ⚡*
    signal Clock6 : Bit_Vector(1 to 2) := (Clock5'range => '1'); -- line 10 ⚡*
    signal Clock7 : Bit_Vector(1 to 1) := ('1'); -- line 11 ⚡*
    signal Clock8 : Bit_Vector(1 to 2) := ('1', '1');
    signal Clock9 : Bit_Vector(1 to 1) := '1'; -- line 13 ⚡*

begin
    Clock1 <= (Clock1'range => '1');
    Clock2 <= (Clock2'range => '1');
end;
```

**Figure 2.4-1 Array Initialization (arrays\aryinit.vhd)**

vcom C:/BOOK/DISK2/ELEMENT/ARYINIT.VHD ←	Model Technology compilation transcript
...	
# ERROR: ARYINIT.VHD(9): Unknown identifier: clock5	
# ERROR: ARYINIT.VHD(9): Attribute range requires an array prefix	
# ERROR: ARYINIT.VHD(9): Type conflict in attribute expression. Type (None) versus natural.	
# ERROR: ARYINIT.VHD(10): Aggregate length is 1. Expected length is 2.	
# ERROR: ARYINIT.VHD(11): type error in enumeration literal: '1'. Expected type Bit_Vector	
# ERROR: ARYINIT.VHD(13): type error in enumeration literal: '1'. Expected type Bit_Vector	

**A**

It is illegal to refer to an identifier within its own declaration. In other words, one cannot refer to the name "clock5" during the declaration of "clock5." The name clock5 is not usable until the end of the declaration (i.e., the ';'). Clock6 fails because the aggregate length is incorrect. This can be corrected as shown below:

```
constant c5 : Bit_Vector(1 to 1) := "1";
signal clock5 : Bit_Vector(1 to 1) := (c5'range => '1'); -- ☺
signal clock5 : Bit_Vector(1 to 1) := "1"; -- ☺
signal clock6 : Bit_Vector(1 to 2) := (others => '1'); -- ☺
```

It is also illegal to initialize a signal of type *Bit\_Vector* with a value of type *bit* (as shown for Clock7 and Clock9). If it is desired to use an aggregate of length one, then the use positional notation is necessary. Otherwise, the compiler interprets the ('1') as a single element (e.g., *Bit*) rather than an aggregate. For example:

```

signal clock7 : Bit_Vector(1 to 1) := (others => '1'); -- ☺
signal clock7 : Bit_Vector(1 to 1) := (1 => '1'); -- ☺

```

A string literal notation can also be used as follows:

```
signal clock9 : Bit_Vector(1 to 1) := "1"; -- note the double quote. ☺
```

Figure 2.4-2 demonstrates another example of illegal referencing of an identifier until the interface list is complete. Potential solutions include:

- Use an unconstrained array for the generic. Array is constrained either with an initialization value, and/or with a value upon component instantiation or in a configuration.
- Use a constant or deferred constant, declared in a package, to bind the generic *B* array size.

```

entity G2 is
  generic(A : Integer := 4;
          B : Bit_Vector(A - 1 downto 0)); -- ⚡
end G2;

entity G is -- OK
  generic(A : Integer := 4;
          B : Bit_Vector := "1011"); -- ☺
end G;

library Work;
  use Work.C_Pkg;
entity G2 is
  generic(A : Integer := 4;
          B : Bit_Vector(C_Pkg.Size_c - 1 downto 0));
end G2;

```

Array is constrained either with an initialization value, and/or with a value upon instantiation, or in a configuration.

The statement "use Library\_Name.Package\_Name" allows the referencing of objects or types defined in the package without the library name. This approach enhances readability.

Figure 2.4-2 Illegal Referencing of an Identifier (arrays\generic.vhd)

Another example of an illegal referencing to an identifier in its own declaration is shown in figure 2.4-3.

```
entity XX is
port (A: in Bit_vector (0 to 4);
      B: in Bit_vector (0 to A'right));
end XX;
```

“Cannot reference A until the interface list is complete.”

Figure 2.4-3. Illegal Identifier Referencing

Section 10.2.3 addresses the issues of parameterization. A recommended solution is to use a package to describe the subtypes for the intended ranges as shown in figure 2.4-4.

```
package Size_Pkg is
  constant Asize_c : Integer := 4;
  subtype A_Typ is Integer range 0 to Asize_c;
end Size_Pkg;

library Work;
use Work.Size_Pkg.all;

entity XY is
port (A: in Bit_Vector(A_Typ); -- from Size_Pkg
      B: in Bit_Vector(A_Typ));
end XY;
```

Subtype for range is defined in a package and used in the object definition.

Figure 2.4-4 Use of a package to Identify Types used in Ports

## 2.5 CONSTANT ARRAYS IN CASE

**Q**

Why does the code shown in figure 2.5-1 yield an error?

```

Library IEEE;
use IEEE.Std_Logic_1164.all;

entity CaseConstant is
  generic(Depth_g : Integer := 4);
end CaseConstant;

architecture CaseConstant_a of
  CaseConstant is
    type AddrState_Typ is array(0 to Depth_g) of
      Std_Logic_Vector(5 downto 0);
    constant Gray : AddrState_Typ := ("000000",
                                         "000001",
                                         "000011",
                                         "000010");
    constant Gray0 : Std_Logic_Vector(5 downto 0) := "000000";
    signal Wa_s : Std_Logic_Vector(5 downto 0) := "000000";

begin
  Test_Lbl: process
  begin
    case wa_s is
      when Gray(0)      => Wa_s <= Gray(1); -- this line fails ☹
      when Gray0        => Wa_s <= Gray(1); -- this line passes ☺
      when others       => Wa_s <= Gray(0);
    end case;
    wait;
  end process Test_Lbl;
end CaseConstant_a;

```

Objects of this type are NOT locally static, but globally static because of the use of the generic which can be modified, and evaluated at elaboration.

Case choice must be locally static

Figure 2.5-1 Case statement with Error (synths\casecst2.vhd)

**A**

The case choice must be locally static. Locally static expressions [1] are expressions evaluated during analysis of the design unit in which they appear, such as

constants of an architecture. There are two other kinds of expressions that are evaluated at different phases of the VHDL compilation and simulation processes. These include globally static expressions [1] are expressions which are evaluated as soon as the design hierarchy in which they appear is elaborated, such as constants that are functions of generics, or generic expressions. The last kinds of expressions are dynamic expressions that are evaluated during initialization or simulation of an architecture. Dynamic expressions include variables, or constants of subprograms, initialized to either values or attributes of formal parameters of the subprogram. Figure 2.5-2 demonstrates the use of a locally static expression.

```

library IEEE;
use IEEE.Std_Lock_1164.all;
entity CaseConstant is
end CaseConstant;

architecture CaseConstant_a of CaseConstant is
type AddrState_Typ is array(0 to 3) of Std_Lock_Vector(5 downto 0);
-- type AddrState_Typ is array(Natural range<>)
--   of Std_Lock_Vector(5 downto 0); -- OK too
--   -- (arrays\casecst.vhd)
constant Gray : AddrState_Typ := ("000000", -- Locally static
                                    "000001",
                                    "000011",
                                    "000010");

constant Gray0 : Std_Lock_Vector(5 downto 0) := "000000";
signal Wa_s : Std_Lock_Vector(5 downto 0) := "000000";
begin
Test_Lbl: process
begin
  case wa_s is
    when Gray(0)      => Wa_s <= Gray(1);      -- ☺
    when Gray0        => Wa_s <= Gray(1);      -- this line passes
    when others       => Wa_s <= Gray(0);
  end case;
  wait;
end process Test_Lbl;
end CaseConstant_a;

```

**Figure 2.5-2 Case statement with Locally Static Expression  
(arrays\casecst.vhd)**

## 2.6 CONSTRAINED AND UNCONSTRAINED ARRAYS

**Q** | What are the differences between constrained and unconstrained arrays? What are the constraining methods? What objects can be constrained or unconstrained?

**A** | [1] An array type is 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. An entire array is referenced with a single identifier (e.g., *Array\_Name*). An individual component of the array is referenced with the array identifier, followed by an index value (or sequence of indexes) placed in parentheses (e.g., *Array\_Name*(4) or *Array\_Name*(4,6)). Arrays are categorized as constrained or unconstrained. A constrained array is an array whose size is restricted (e.g., 10 elements). An unconstrained array is an array that defines the type of elements, but not its size. This allows users to declare arrays that differ only in size (the number of index value in a given dimension) to be of the same type. An unconstrained array does not include information about the size of the array. For example, the following type represents an unconstrained array of *Std\_Lock* as defined in package *IEEE.Std\_Lock\_1164*:

```
type Std_Lock_Vector is array ( Natural range <>) of Std_Lock;
```

In that case, the index type of the unconstrained array is *Natural*, a subtype of *Integer* type. Note that if unconstrained arrays were not in the language, then one would have to define a new type for every array length, and those types would not be compatible. For example, figure 2.6 demonstrates the problem of not using unconstrained arrays.

```

architecture U_Ary_a of U_Ary is
  type MyBit_Typ is ('0', '1');
  type Mbit16_Typ is array(15 downto 0) of MyBit_Typ;
  type Mbit32_Typ is array(31 downto 0) of MyBit_Typ;
begin
  Test_Lbl: process
    variable M16_v : Mbit16_Typ;
    variable M32_v : Mbit32_Typ;
  begin
    M32_v(31 downto 16) := M16_v; -- ERROR, INCOMPATIBLE TYPES ⚡*
    wait;
  end process Test_Lbl;
  Test2_Lbl: process
    variable M16_v : Bit_Vector(15 downto 0);
    variable M32_v : Bit_Vector(31 downto 0);
  begin
    M32_v(31 downto 16) := M16_v; -- OK, Same base type 😊
    wait;
  end process Test2_Lbl;
end U_Ary_a;

```

M16\_v and M32\_v are of different types, and slices of equal length from each variable are incompatible.

M16\_v and M32\_v are of same type, and slices of equal length from each variable are compatible.

**Figure 2.6 Unconstrained and Constrained Array Restrictions  
(arrays\ary.vhd)**

### 2.6.1 Constraining Methods

An array can be constrained using any of the following methods:

1. Indexed constraint Direct value  
signal S : Std\_Logic\_Vector( 15 downto 0); -- locally static
2. Subtype definition Subtype  
subtype Int32\_Typ is Integer range 31 downto 0;  
subtype Stda32\_Typ is Std\_Logic\_Vector(31 downto 0); -- locally static  
subtype Std32\_Typ is Std\_Logic\_Vector(Int32\_Typ); -- locally static
3. Use of a generic Generic  
entity X is  
 generic(Width\_g : Natural := 8);  
 port(A : out Bit\_Vector(Width\_g-1 downto 0)); -- globally static

4. Use of a constant declared in a package, entity, or architecture

```
constant Width_c : natural := 8;
```

Constant

```
signal S : Bit_Vector(Width_c -1 downto 0); -- locally static
```

5. Use of Attributes

```
entity Y is
    port(B : out Bit_Vector);
end Y;
```

Attribute. Actual length of vector is defined at either component instantiation (for component) or at subprogram call (for procedures or functions)

```
architecture Y_a of Y is
    signal S : Bit_Vector(B'length - 1 downto 0); -- globally
                                                    -- static
```

## 2.6.2 Allowed Constrained and Unconstrained Objects

Table 2.6.2 summarizes the category of objects that are allowed as constrained and unconstrained arrays. It also provides information about what many synthesizers can and cannot accept.

Table 2.6.2 Allowed Constrained and Unconstrained Objects

CATEGORY	CONSTRAINED ARRAY	UNCONSTRAINED ARRAY
Generic	- Any type for VHDL - May be limited to Integer type for synthesis	- Any type for VHDL - May be limited to Integer type for synthesis
Constant	- Any type for VHDL	Constants are constrained by the value (e.g. <b>constant C : Bit_Vector := "1010";</b> ). Deferred constants constrained in package body (deferred constants are illegal in synthesis).
Port	- Any type for VHDL - For synthesis , one dimensional array of <i>Bit</i> , <i>Std_Logic</i> , or <i>Boolean</i> type	- Any type for VHDL - For synthesis , one dimensional array of <i>Bit</i> , <i>Std_Logic</i> , <i>Integer</i> , or <i>Boolean</i> type
Signal	- Any type for VHDL - Must be constrained	- Illegal
Variable	- Any type for VHDL - Must be constrained	- Illegal
Function	- constraining size of the array may cause loss of re-usability, but may be desired to fix functionality.	Good practice to use unconstrained arrays for parameters and return discrete or unconstrained array types. Use attributes to constrain subprogram local variables
Procedure	- constraining size of the array may cause loss of re-usability, but may be desired to fix functionality	Good practice to use unconstrained arrays for parameters. Use attributes to constrain subprogram local variables

In synthesis, two dimensional arrays are generally not allowed for ports, signals, and variables. They are allowed as constants for use as lookup tables. A one-dimensional array is restricted to an array of objects of type *Bit*, *Bit\_Vector*, *Std\_Logic*, *Std\_Logic\_Vector*, *Std\_uLogic*, *Std\_uLogic\_Vector*, *integer* (and *integer* subtypes), and *Boolean*.

## 2.7 MAPPING ARRAYS OF DIFFERENT SIZES

**Q**

Given the following array types:

```
subtype A5_Typ      is Std_Logic_Vector(4 downto 0);
subtype A3_Typ      is Std_Logic_Vector(2 downto 0);
type   Mem5x5_Typ is array(4 downto 0) of A5_Typ;
type   Mem3x3_Typ is array(2 downto 0) of A3_Typ;
type   A2D5x5_Typ is array(4 downto 0, 4 downto 0)
                  of Std_Logic;
```

1. How can an object of *A3\_Typ* type be mapped into an object of *A5\_Typ* type?
2. How can an object of *Mem3x3\_Typ* type be mapped into an object of *Mem5x5\_Typ* type?
3. How can an object of *Mem3x3\_Typ* type be mapped into an object of *A2D5x5\_Typ* type?
4. How can values be assigned to array aggregates?

**A**

For one dimensional array objects of type *Bit\_Vector*, *Std\_Logic\_Vector*, *Signed*, or *Unsigned*, either array slices or loops can be used to map one array slice onto another. For multidimensional arrays of different sizes, loops are required to map the individual slices or elements of the arrays. Figure 2.7 demonstrates an example for the mapping of arrays of different sizes.

```
... -- empty entity
architecture array_of_array of MultiDM is
begin
  subtype A5_Typ      is Std_Logic_Vector(4 downto 0);
  type   Mem5x5_Typ is array(0 to 4) of A5_Typ;
  type   A2D5x5_Typ is array(0 to 4, 0 to 4) of Std_Logic;

  subtype A3_Typ      is Std_Logic_Vector(2 downto 0);
  type   Mem3x3_Typ is array(0 to 2) of A3_Typ;
  type   A2D3x3_Typ is array(0 to 2, 0 to 2) of Std_Logic;

  signal A5_s      : A5_Typ      := "01011";
  signal Mem5x5_s : Mem5x5_Typ  -- array(0 to 4) of A5_Typ;
  := (0 => "10011",
      1 => (others => '1'),    ← Aggregate
      2 => "11010",
      3 => (others => 'H'),
      4 => "Z01LL");

  signal A2D5x5_s : A2D5x5_Typ  -- array(0 to 4, 0 to 4) of Std_Logic;
  := (0 => ('U', 'X', '0', '1', 'Z'),
      1 => ('W', 'L', 'H', '-', '1'),
      2 => ('1', 'U', '0', 'Z', 'W'),    ← Aggregate
      3 => ('0', '1', '0', 'X', '0'),
      4 => ('H', '0', 'L', '0', 'W'));

  signal A3_s      : A3_Typ;
  signal Mem3x3_s : Mem3x3_Typ;
  signal A2D3x3_s : A2D3x3_Typ;
```

```

begin
Test_Lbl: process
begin
  A3_s <= "HLX"; -- simple assignment
  A5_s <= "1-HLZ";
  wait for 10 ns;

  A5_s(4 downto 2) <= A3_s; -- Assignment of a slice
  Mem3x3_s(2) <= A3_s;
  A2D3x3_s(2,2) <= A5_s(0); -- One element is updated

  Mem5x5_s <= (0 => (A2D5x5_s(0,0) &
                        A2D5x5_s(0,1) &
                        A5_s(3 downto 1)),
                1 => (A3_s & A5_s(1 downto 0)), -- 5 bits total
                2 => A5_s,
                3 => (others => 'Z'),
                4 => "0LZU1"); -- Update of whole aggregate
  wait for 10 ns;

-- How can an object of Mem3x3_Typ be mapped into an object of Mem5X5_Typ?
-- Mem3x3_s <= Mem5x5_s(2 to 4) (4 downto 2) -- illegal
LP1_Lbl: for I in 0 to 2 loop
  Mem3x3_s(I) <= Mem5x5_s(I + 2)(4 downto 2);
end loop LP1_Lbl;

wait for 10 ns;

-- How can an object of Mem3x3_Typ be mapped into an object of A2D5x5_Typ?
-- Mem3x3_s <= A2D5x5_s(2 to 4, 2 to 4) - Illegal
LP2_Lbl: for I in 0 to 2 loop
  Lp3_Lbl: for J in 0 to 2 loop
    Mem3x3_s(I)(J) <= A2D5x5_s(I + 2, 4 - J);
  end loop Lp3_Lbl;
end loop LP2_Lbl;
wait for 10 ns;
end process Test_Lbl;
end array_of_array;

```

**Figure 2.7 Mapping of Arrays of Different Sizes (arrays\multidm.vhd)**

## 2.8 UNCONSTRAINED AGGREGATE WITH "OTHERS "



Why does the concurrent assignment in figure 2.8-1 give this error? “ An unconstrained array aggregate may not have an others association”

```

entity Test is
  port(A : out Bit_Vector);
end;
architecture Versl of Test is
  procedure P(X : out String) is
    begin
      X := (others => 'x'); -- ⚡
      -- # ERROR: 'Others' is in unconstrained array aggregate
    end P;

```

```

begin
  A <= (others => '0');  -- ♦*
  -- # ERROR: 'Others' is in unconstrained array aggregate
end Vers1;

```

**Figure 2.8-1 Constrained Aggregate with "others", (arrays\uncons.vhd)**

**A** LRM 7.3.2.2 lists all the contexts in which *others* may appear in an array aggregate. This section states that *others* may appear in an array aggregate when the aggregate's bounds are determined by some constrained array subtype. In the above example, port *A* is of type *Bit\_Vector*, an unconstrained array type.

There is no inherent reason for this restriction. By elaboration time, the subtype of port *A* will have been constrained by an actual in some port map. By execution time the range of the aggregate containing *others* will be known. Ada, from which VHDL was derived, allows a similar construct with the procedure. The Ada rules recognize that a constraint may be set at run-time. However, VHDL uses the syntactic term "constrained array subtype" that recognizes constraints defined before run-time. This aggregate rule is an issue that should be revisited in the next version of the language.

The *others* can be used, if the size of the unconstrained array is handled explicitly by using a generic as shown in figure 2.8-2. For the case of a subprogram, a local variable can be constrained with the attributes '*length*' or '*range*'.

```

entity Test is
  generic (N_g : Natural);           ← Generic must be initialized
  port(A : out Bit_Vector(N_g - 1 downto 0)); ← prior to simulation. A default
end;                                     value could be inserted here.
                                         generic (N_g : Natural := 8);

architecture Vers1 of Test is
  procedure P(X : out String) is
    variable X_v : String(1 to X'length);
    begin
      X_v := (others => 'x');
      X := X_v;
    end P;

begin
  A <= (others => '0');  ← others is allowed because port A is constrained.
end Vers1;

```

Local variable is constrained with the *length* attribute

**Figure 2.8-2 Constrained Aggregate with "others", (arrays\unconsok.vhd)**

Another case where the *others* is not allowed in synthesis (with *Synopsys dc\_shell*) is a slice assigned with the *others* construct. This is not an error in the language, but an error during the elaboration process with *Synopsys*. This error is demonstrated in figure 2.8-3. The following assignment yields an error, but the type of the aggregate can be determined from the context. This appears to be an ambiguity in *Synopsys* where explicit type definitions are required.

S2(7 downto 4) <= (others => '0');

There are two solutions to resolve this issue, as shown in figure 2.8-3:

1. Use a type qualifier
2. Use a constant

```

library IEEE;
use IEEE.Std_Logic_1164.all;

entity T is
  generic(Size_G : Integer := 8);
  port (A : in    Std_Logic_Vector(Size_g - 1 downto 0);
        B : out   Std_Logic_Vector(Size_g - 1 downto 0);
        C : out   Std_Logic_Vector(Size_g - 1 downto 0));
end T;

architecture T_a of T is
  constant Zero4_c : Std_Logic_Vector(Size_g - 1 downto 4) := (others => '0');
  subtype Std4_Typ is Std_Logic_Vector(Size_g - 1 downto 4);

  signal S1 : Std_Logic_Vector(Size_g - 1 downto 0);
  signal S2 : Std_Logic_Vector(Size_g - 1 downto 0);

begin -- T_a
  S1(Size_g - 1 downto 4) <= Zero4_c;           ← Fix with typed constant ⚡⚡
  S1(3 downto 0) <= A(3 downto 0);
  B <= S1;

  -- S2(Size_g - 1 downto 4) <= (others => '0');           ← Error in Synopsys,
  S2(Size_g - 1 downto 4) <= Std4_Typ'(others => '0');   ← but NOT in VHDL
  S2(3 downto 0) <= A(3 downto 0);                 ← Fix with type qualifier ⚡⚡
  C <= S2;
end T_a;

```

Figure 2.8-3 Assignment of a Slice with Synopsys (arrays\SynopOk.vhd)

## 2.9 ILLEGAL ARRAY TYPES

**Q** | What array types are illegal in VHDL?

**A** | An anonymous array is an array that does not belong to any named type. Each anonymous array is considered to be one of a kind, that is defined in the declaration of an object. For example:

`variable X : array(1 to 10) of Integer; -- is an anonymous array`  
**ANONYMOUS ARRAYS are ILLEGAL in VHDL.** (They are legal in Ada.) Figure 2.9 demonstrates an example of an anonymous array.

```
subtype RegSize_Typ is integer range 1 to 4;
type aInt_Typ  is array(RegSize_Typ) of integer;
...
variable aInt_v      : aInt_Typ;  ← -- belongs to a define type or subtype
                                     ↓
variable Anonymous_v : array(RegSize_Typ) of integer; -- ⚡*
                                         -- Anonymous Array
```

Figure 2.9 Example of an Illegal Anonymous Array (arrays\anony\_ea.vhd)

VHDL and Ada do not allow a composite type to contain an unconstrained type. The following two declarations are illegal:

```
type Token_Typ is record
    Data : Std_Logic_Vector; -- ⚡* Needs size constraint,
                           -- e.g. Std_Logic_Vector(31 downto 0)
end record;
type VectorOfBitVector_Typ is array(Natural range <>) of Bit_Vector; -- ⚡*
-- Needs size constraint on Bit_Vector, e.g. Bit_Vector(31 downto 0)
```

### 2.9.1 UNCONSTRAINED ARRAY OF AN UNCONSTRAINED ARRAY

**Q** | How can a structure equivalent to an unconstrained array of an unconstrained array be created? For example, the following is illegal:

```
type AddrState_Typ is array(Natural range <>) of -- ⚡*
  std_Logic_Vector(Natural range <>);
```

**A** | The language does not allow composites (arrays, and records) of an unconstrained type. The code generation for indexing and selection needs the size of the elements (this rule also applies in Ada). Some alternative data structures include the following:

- One two-dimensional, unconstrained array, e.g.

```
type AddrState_Typ is array(Natural range <>, Natural range <>) of Std_Logic;
```

It is difficult to map objects of type *Std\_Logic\_Vector* onto a two-dimensional array. The mapping must be performed on a bit by bit basis. This can be done through some user-defined procedures and functions.

- Array range derived from constrained types or from subtypes. This is often the case because the range is a function of the range of other types. For example, in figure 2.9.1-1:

```
package TestTypes_Pkg is
    subtype BV8_Typ is Bit_Vector(7 downto 0);
    type IntAry_Typ is array(0 to 255) of Integer;
    Type Mem_Typ is array(IntAry_Typ'range) of
        Bit_Vector(BV8_Typ'range); -- &amp;
    subtype S256_Typ is Integer range 0 to 255;
    type Mem256_Typ is array(S256_Typ) of BV8_Typ; -- &amp;
    constant Size_c : Natural; -- deferred constant
end TestTypes_Pkg;
```

**Figure 2.9.1-1 Constraining Array Size Based on Size of Other Types  
(arrays\test.vhd)**

- Array range defined by a generic or a Constant. The generic is declared in the entity. The constant can be declared in the entity, architecture, or a package. Deferred constants can be used. Deferred constants are non-synthesizable. The use of a generic to constrain the size of arrays is demonstrated in figure 2.9.1-2:

```
package body TestTypes_Pkg is
    constant Size_c : Natural := 128;
end TestTypes_Pkg;

library Work;
entity Test is
    generic(Depth_g : Natural := 256;
            Width_g : Natural := 8);
end Test;

architecture Test_a of Test is
    type Memory_Typ is array(0 to Depth_g - 1) of
        Bit_Vector(Width_g - 1 downto 0); -- &amp;
    type A_Typ is array(0 to Work.TestTypes_Pkg.Size_c - 1) of
        Bit_Vector(Width_g - 1 downto 0); -- &amp;
begin
end Test_a;
```

**Figure 2.9.1-2 Constraining Array Size Based on Generics and Deferred Constants (arrays\test.vhd)**

### **3. DRIVERS**

---

---

This section addresses issues of unexpected multiple driver errors often experienced by VHDL users. It also addresses the issue of atomicity on drivers and sensitivity of resolved composites.

### 3.1 MULTIPLE DRIVERS - CASE 1

**Q**

Where is the multiple driver error in the following code? Figure 3.1 demonstrates useless code that does not elaborate in *Cadence® Leapfrog*. It refers to LRM 4.3.1.2 to reject it. The displayed error is:

\*> cv drvbit.vhd

cv: v2.2.(6): (c) Copyright 1992-1995, Cadence Design Systems, Inc.

\*> ev -COMPATIBILITY -SNAPSHOT SIM work.TOP:RTL

ev: v2.2.(s1): (c) Copyright 1992-1995, Cadence Design Systems, Inc.

ev: \*E,1122: multiple sources for unresolved signal: A [4.3.1.2].

Building driver for signal /RTL.A from process: /RTL.ARRAY\_LBL

```
entity Top is
end Top;

architecture RTL of Top is
    signal A : Bit_Vector(1 downto 0);
    signal X : Bit;
begin
    Bit_Lbl: process(X)
    begin
        A(0) <= '1';
    end process Bit_Lbl;
    A(0) has one driver
    Array_Lbl: process(X)
    begin
        for I in 1 to 1 loop -- NO PROBLEM WITHOUT THE LOOP AND A(1)
            A(I) <= '0'; -- ⚡
        end loop;
    end process Array_Lbl;
    One driver for each element of A
    because A(I) is NOT STATIC
end RTL;
```

**Figure 3.1 Example Demonstrating Problem with Drivers  
(drivers\drvbit.vhd)**

**A**

The first process (*Bit\_Lbl*) has a driver for only *A(0)*. The second process (*Array\_Lbl*) has a driver for the entire signal *A* because *A(I)* is not a static name.

*A(0)* has multiple drivers and signal *A* is not resolved, hence the error. If *A(I)* is explicitly used in the second process, then that process drives only *A(I)*, and the multiple drivers condition would not exist. The misconception is that a smart analyzer can recognize the loop index as just one, and can create a single driver on *A(I)*. Actually, drivers are created during static elaboration. The loop index is not elaborated until execution of the loop. At elaboration, drivers are created for every element of the array because the simulator has no knowledge of the loop index range. It must then assume that the index can take any of the allowed values. A similar condition holds for the *if* statement. For example, the statement *if GenericValue then S1 <= '1'; else S2 <= '0'; end if;* causes a driver on S1 and S2.

Subsection 3.1.1 provides a *Std\_Logic\_Vector* solution. Subsection 3.2.2 provides a separate signals solution. Subsection 3.2.3 discusses the issue of atomicity and LRM 4.3.1.2 that addresses the effects of drivers and resolved signals.

### 3.1.1 Std\_Logic\_Vector Solution

Changing the types of the signals from unresolved *bit* to resolved *Std\_Logic\_Vector* helps in avoiding the runtime error, but does not avoid the drivers for each element of *A* in process *Array\_Lbl* because *A(I)* is NOT STATIC. Specifically, this process drives *A(I)* to '0', and drives *A(0)* to the default value of signal *A(0)*, or a 'U' (since the signal is uninitialized). This type of assignment is defined in LRM 12.6.1 that states:

[1] *Each signal assignment is associated with a driver. The initial contents of a driver associated with a given signal are defined by the default value associated with the 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 affects only the associated driver(s).*

This problem is demonstrated in figure 3.1.1. To solve this problem, either signal *A* needs to be initialized to a "ZZ" in the signal declaration, or process *Bit\_Lbl* needs to drive *A(I)* to a 'Z'. Note that synthesis ignores initialization values, causing a potential mismatch between synthesized logic and simulated logic.

```
Library IEEE;
use IEEE.Std_Logic_1164.all;
entity TopStd is
end TopStd;

architecture RTL of TopStd is
    signal A : Std_Logic_Vector(1 downto 0); -- initialized to "UU"
    signal X : Std_Logic;
begin
    Bit_Lbl: process(X)
    begin
        A(0) <= '1';
    end process Bit_Lbl;
    Array_Lbl: process(X)
    begin
        for I in 1 to 1 loop
            A(I) <= '0';
        end loop;
    end process Array_Lbl;
end RTL;
```

Since type *Std\_Logic\_Vector* is defined as an unconstrained array of resolved *Std\_Logic*, the assignment to *A(0)* creates ONE driver for *A(0)* only.

Since the loop index can be of any range, drivers are created for every element of the array. These drivers are initialized to the default value of *A* (a "UU").

Simulation Results
ns      delta      A      X
0      +0      UU      U
0      +1      OU      U

**Figure 3.1.1 Example Demonstrating Problems with Drivers  
(drivers\drvstd.vhd)**

### 3.1.2 Separate Signals Solution

One solution is to create separate signals for each of the drivers. A concurrent signal assignment can then be used to concatenate those individual signals onto a single vector (shown in figure 3.1.2). This solution might however lead to additional buffers created by the synthesizer.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

entity TopStd2 is
end TopStd2;

architecture RTL of TopStd2 is
    signal A : Std_Logic_Vector(1 downto 0);
    signal B : Std_Logic_Vector(1 downto 0);
    signal A0 : Std_Logic;
    signal A1 : Std_Logic;
    signal X : Std_Logic;
begin
    Bit_Lbl: process(X)
    begin
        A0 <= '1';
    end process Bit_Lbl;

    Array_Lbl: process(X)
    begin
        A1 <= '0';
    end process Array_Lbl;

    A <= (A1, A0); -- Option 1
    B <= A1 & A0; -- Option 2
end RTL;

```

Simulation Results

ns	delta	a	b	a0	a1	x
0	+0	UU	UU	U	U	U
0	+1	UU	UU	1	0	U
0	+2	01	01	1	0	U

Aggregate

Expression

**Figure 3.1.2 Separate Signals Resolution to Avoid Multiple Drivers  
(drivers\drvstd2.vhd)**

### 3.1.3 Atomicity

There is a misinterpretation of paragraph 4.3.1.2 of the LRM which states: *[1] 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.* That statement is true for resolved signals but is NOT true for an unresolved bit array.

To understand this concept, figure 3.1.3-1 demonstrates two type definitions:

1. *Std16u\_Typ* represents an array 15 down to 0 of *Std\_uLogic* elements.
2. *Std16\_Typ* represents a resolved *Std16u\_Typ*.

Note that type *Std16\_Typ* looks very similar to *Std\_Logic\_Vector(15 downto 0)* but it is resolved differently. An object of type *Std\_Logic\_Vector* represents an array of individual elements of type *Std\_Logic*, and each element is resolved SEPARATELY. An object of *Std16\_Typ* is resolved as an atomic element (e.g., as a whole, or in mass).

There are two important differences resulting from the distinction between an array of atomic elements (resolved individually, e.g., *Std\_Logic\_Vector*) and an atomic element (a fixed array resolved as a whole).

1. **DRIVERS:** LRM 4.3.1.2, [1] If a subelement of a resolved signal of composite type (e.g. array or record, like *Std16\_Typ*) 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 those drivers taken together constitute one source of the signal. LRM 12.6.2 states that [1] 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 *S* is equal to the aggregate of driving values of each of the basic signals that are the subelements of *S*. Thus, if signal *R* is a resolved signal, then a signal assignment on *R(0)* causes a driver for every element of *R*. The value driven on the other elements (e.g. *R(1 to 15)*) will be the default value of the corresponding elements of *R* (e.g., the default values of *R(1 to 15)* defined in the signal declaration). LRM 4.3.1.2, [1] 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*.

```
-- subtype Std16u_Typ is Std_uLogic_Vector(15 downto 0);
-- Type Std16uAry_Typ is array(natural range<>) of Std16u_Typ;
-- function Resolved(Drvs : Std16uAry_Typ) return Std16u_Typ;
-- subtype Std16_Typ is resolved Std16u_Typ;
signal R : Std16_Typ := (others => 'Z'); -- Resolved at atomic level
-- Initialized to "ZZZZZZZZZZZZZZZZZZ"
signal Q : Std16_Typ; -- Resolved at atomic level
-- Initialized to "UUUUUUUUUUUUUUUU"
signal S : Std_Logic_Vector(15 downto 0);
-- Array of individually resolved signals
...
R(0) <= '1'; -- causes a driver for every element of R
-- Thus, R(15 downto 1) drive the value "ZZZZZZZZZZZZZZZZ"
-- R(0) drives '1'

Q(0) <= '1'; -- causes a driver for every element of Q
-- Thus, Q(15 downto 1) drive the value "UUUUUUUUUUUUUUUU"
-- Q(0) drives '1'

S(0) <= '1'; -- Causes ONE and only ONE driver on S(0) with a value of '1'
-- There are no drivers on S(15 downto 1)
```

2. **ACTIVITY:** If a subelement of a resolved signal of composite type has a driver, then an activity on any one element causes an activity on the whole composite. LRM 12.6.2 states that [1] a signal is said to be active during a given simulation cycle if (one of the conditions) the signal is a subelement of a resolved signal and the resolved signal is active. In the following example, any assignment on *R* (e.g. *R(15)*) causes this process to fire because there is an activity on all the bits of *R*. This process is sensitive to an activity ('transaction) on the signal *R(0)*. This is NOT the case if *R* is unresolved, or if *R* is an array of resolved elements (such as *Std\_Logic\_Vector* that is an array of *Std\_Logic* elements).

```

signal R : Std16_Typ := (others => 'H'); -- Resolved type at the atomic level
Test_Lbl: process(R(0)'transaction) -- Process sensitive on ANY activity of R
-- including R(15), R(14), ...or R(1) or R(0)

```

A composite signal is equivalent to N scalar signals, each with its own future waveform, last value, last event, activity flag, etc. Therefore, a sensitivity on the entire bus is equivalent to the sensitivity on each bit of the bus. A process with a sensitivity clause triggers on events that represent a change in the value of the signals defined in the sensitivity clause. To make a process sensitive to activities on signals, it is necessary to make the process sensitive to the *Signal\_Name'transaction*. These concepts of atomicity are demonstrated in figure 3.1.3-1 and 3.1.3-2.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
package Atomic_Pkg is
    subtype Std16u_Typ is Std_Logic_Vector(15 downto 0);
    type Std16uAry_Typ is array(natural range<>) of Std16u_Typ;
    function Resolved(Drvs : Std16uAry_Typ) return Std16u_Typ;
    subtype Std16_Typ is resolved Std16u_Typ;
end Atomic_Pkg;

package body Atomic_Pkg is
    type STDLOGIC_TABLE is array(STD_ULOGIC, STD_ULOGIC) of STD_ULOGIC;
    constant RESOLUTION_TABLE : STDLOGIC_TABLE := (
        -- copied from std_logic_1164 package
        --
        --   | U   X   0   1   Z   W   L   H   -   |   |
        --   | 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'  |   | U |
        --   | 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'  |   | X |
        --   | 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X'  |   | 0 |
        --   | 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X'  |   | 1 |
        --   | 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'  |   | Z |
        --   | 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'  |   | W |
        --   | 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'  |   | L |
        --   | 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'  |   | H |
        --   | 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'  |   | - |
    );
    function Resolved(Drvs : Std16uAry_Typ) return Std16u_Typ is
        -- weakest state default.
        variable Result : Std16u_Typ := (others => 'Z');
    begin
        if Drvs'length = 1 then -- check for only one driver.
            return Drvs(Drvs'low); -- return 1st input
        end if;
        -- loop through all the Drivers on the bus.
        Drv_LBL: for Drv_i in DRVS'range loop
            -- Loop through all bits
            Bit_Lbl: for Bit_i in Result'range loop
                RESULT(Bit_i) := RESOLUTION_TABLE(RESULT(Bit_i),
                                                DRVS(Drv_i)(Bit_i));
            end loop Bit_Lbl;
        end loop Drv_Lbl;
        return RESULT;
    end Resolved;
end Atomic_Pkg;

```

**Figure 3.1.3-1 Package Demonstrating Resolved Array Type  
(drivers\atomic\_pb.vhd)**

```

library IEEE;
  use IEEE.Std_Logic_1164.all;
library Work;
  use Work.Atomic_Pkg.all;
entity DrvAtomic is
end DrvAtomic;

architecture DrvAtomic_a of DrvAtomic is
  signal R : Std16_Typ := (others => 'H'); -- Resolved type
begin
  A1_Lbl: process(R)  -- sensitive to ALL bits, 15 downto 0
  begin
    R(15) <= 'Z';           ←
    assert False
      report "A1_lbl wakes up"
      severity note;
    end process A1_Lbl;

  B1_Lbl: process(R(0)) -- sensitive to R(0)
  begin
    R(8) <= 'Z';           ←
    assert False
      report "B1_lbl wakes up"
      severity note;
    end process B1_Lbl;

  C1_Lbl: process(R(7 downto 0), R(15))
    -- sensitive to R(7 downto 0) and R(15)
  begin
    R(9) <= 'Z';           ←
    assert False
      report "C1_lbl wakes up"
      severity note;
    end process C1_Lbl;

  R(15) <= 'Z' after 2 ns,
          '0' after 10 ns,
          'Z' after 20 ns,
          'Z' after 30 ns,
          '1' after 40 ns,
          'Z' after 50 ns,
          'Z' after 60 ns;
end DrvAtomic_a;

```

Drivers for ALL bits of *R* when *R* is resolved at the atomic level. One driver for *Std\_Logic\_Vector* that represents an array of individually resolved signals.

**Figure 3.1.3-2 Process Sensitivity on Resolved Signal (drivers\atmdrv5.vhd)**

File *drivers\drvstd5.vhd* is the same as *drivers\atmdrv5.vhd* except that signal *R* is defined as shown below:

```

subtype StdLogic16_Typ is Std_Logic_Vector(15 downto 0);
signal R: StdLogic16_Typ := (others => 'H'); -- Array of individually resolved type

```

Figure 3.1.3-3 compares the simulation results of drivers of a resolved signal (as a whole) versus an array of resolved signals (*Std\_Logic\_Vector*).

RESOLVED SIGNAL (Work.Atomic_Pkg.Std16_Typ)	ARRAY OF RESOLVED SIGNAL (Std_Logic_Vector)
run 10	run 10
# ** Note: C1_lbl wakes up	# ** Note: C1_lbl wakes up
# Time: 0 ns Iteration: 0 Instance:/	# Time: 0 ns Iteration: 0 Instance:/
# ** Note: B1_lbl wakes up	# ** Note: B1_lbl wakes up
# Time: 0 ns Iteration: 0 Instance:/	# Time: 0 ns Iteration: 0 Instance:/
# ** Note: A1_lbl wakes up	# ** Note: A1_lbl wakes up
# Time: 0 ns Iteration: 0 Instance:/	# Time: 0 ns Iteration: 1 Instance:/
# ** Note: A1_lbl wakes up	# ** Note: A1_lbl wakes up
# Time: 2 ns Iteration: 0 Instance:/	# Time: 2 ns Iteration: 0 Instance:/
# ** Note: C1_lbl wakes up	# ** Note: C1_lbl wakes up
# Time: 2 ns Iteration: 0 Instance:/	# Time: 2 ns Iteration: 0 Instance:/
# ** Note: A1_lbl wakes up	# ** Note: A1_lbl wakes up
# Time: 10 ns Iteration: 0 Instance:/	# Time: 10 ns Iteration: 0 Instance:/
# ** Note: C1_lbl wakes up	# ** Note: C1_lbl wakes up
# Time: 10 ns Iteration: 0 Instance:/	# Time: 10 ns Iteration: 0 Instance:/
drivers R(0)	drivers r(0)
# Signal r = H	# Signal r = H
# Driver line_43 = H	NO driver on R(0)
# Driver c1_lbl = H	
# Driver b1_lbl = H	
# Driver a1_lbl = H	
drivers R(15)	drivers r(15)
# Signal r = 0	# Signal r = 0
# Driver line_43 = 0, Z at 20 ns, Z at 30 ns, 1 at	# Driver line_38 = 0, Z at 20 ns, Z at 30 ns,
40 ns, Z at 50 ns, Z at 60 ns	1 at 40 ns, Z at 50 ns, Z at 60 ns
# Driver c1_lbl = H	# Driver a1_lbl = Z
# Driver b1_lbl = H	
# Driver a1_lbl = Z	
-----	-----
- List file	- List file
ns delta R	Drivers on ALL bits from each process with initial default values.
0 +0 HHHHHHHHHHHHHHHHHHH	
10 +0 0HHHHHHHHHHHHHHHHHH	Drivers on bits 15, 8, 9. Other bits are at default values.

**Figure 3.1.3-3 Simulation Results of Drivers of a Resolved Signal Versus an Array of Resolved Signals.**

Figure 3.1.3-4 illustrates the sensitivity on processes with resolved signals. The processes are sensitive to signal '*transaction*'.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

library Work;
use Work.Atomic_Pkg.all;

entity Atomic is
end Atomic;

architecture Atomic_a of Atomic is
-- subtype Std16u_Typ is Std_uLogic_Vector(15 downto 0);
-- Type Std16uAry_Typ is array(natural range<>) of Std16u_Typ;
-- function Resolved(S : Std16uAry_Typ) return Std16u_Typ;
-- subtype Std16_Typ is resolved Std16u_Typ;
signal R : Std16_Typ := (others => 'H'); -- Resolved type

begin

  A1_Lbl: process(R'transaction)
    -- sensitive to ALL bits, 15 downto 0
  begin
    assert False
      report "A1_lbl wakes up"
      severity note;
  end process A1_Lbl;

  B1_Lbl: process(R(0)'transaction)
  begin
    assert False
      report "B1_lbl wakes up"
      severity note;
  end process B1_Lbl;

  C1_Lbl: process(R(7 downto 0)'transaction, R(15)'transaction)
  begin
    assert false
      report "C1_lbl wakes up"
      severity note;
  end process C1_Lbl;

  R(15) <= 'Z' after 2 ns,
           '0' after 10 ns,
           'Z' after 20 ns,
           'Z' after 30 ns,
           '1' after 40 ns,
           'Z' after 50 ns,
           'Z' after 60 ns;
end Atomic_a;

```

Since R is resolved at the "atomic" level, any activity on any bits on R (e.g. R(15)) will cause an activity on the whole signal. Thus, R(0)'transaction is an event whenever there is an activity on R.

**Figure 3.1.3-4 Sensitivity on Processes with Resolved Signal (atomic5.vhd)**

File (drivers\atomstd5.vhd) represents the same code as shown in figure 3.1.3-4 except that signal R is of type *StdLogic16\_Typ* as defined below:

```

subtype StdLogic16_Typ is Std_Logic_Vector(15 downto 0);
signal R : StdLogic16_Typ := (others => 'H');

```

Figure 3.1.3-5 compares the simulation results on sensitivity on activity of signals in processes.

Resolved Composite (Work.Atomic_Pkg.Std16_Typ)	Array of Resolved Signals (Std_Logic_Vector)
<pre> # ** Note: C1_lbl wakes up # Time: 0 ns Iteration: 0 Instance:/ # ** Note: B1_lbl wakes up # Time: 0 ns Iteration: 0 Instance:/ # ** Note: A1_lbl wakes up # Time: 0 ns Iteration: 0 Instance:/  # ** Note: B1_lbl wakes up ✓ Sensitive to ANY # Time: 2 ns Iteration: 0 Instance:/ # ** Note: A1_lbl wakes up # Time: 2 ns Iteration: 0 Instance:/ # ** Note: C1_lbl wakes up # Time: 2 ns Iteration: 0 Instance:/  # ** Note: B1_lbl wakes up # Time: 10 ns Iteration: 0 Instance:/ # ** Note: A1_lbl wakes up # Time: 10 ns Iteration: 0 Instance:/ # ** Note: C1_lbl wakes up # Time: 10 ns Iteration: 0 Instance:/  -- List file ns delta          R 0   +0 HHHHHHHHHHHHHHHHHHH 2   +0 ZHHHHHHHHHHHHHHHHHH 10  +0 OHHHHHHHHHHHHHHHHHH </pre>	<pre> # ** Note: C1_lbl wakes up # Time: 0 ns Iteration: 0 Instance:/ # ** Note: B1_lbl wakes up # Time: 0 ns Iteration: 0 Instance:/ # ** Note: A1_lbl wakes up # Time: 0 ns Iteration: 0 Instance:/  # ** Note: A1_lbl wakes up ✗ Sensitive to ACTIVITY in sensitivity # on specific bits of R (e.g. R, R(15)) # Time: 2 ns Iteration: 0 Instance:/ # ** Note: C1_lbl wakes up # Time: 2 ns Iteration: 0 Instance:/  # ** Note: A1_lbl wakes up # Time: 10 ns Iteration: 0 Instance:/ # ** Note: C1_lbl wakes up # Time: 10 ns Iteration: 0 Instance:/  -- List file ns delta          R 0   +0 HHHHHHHHHHHHHHHHHHH 2   +0 ZHHHHHHHHHHHHHHHHHH 10  +0 OHHHHHHHHHHHHHHHHHH </pre>

Figure 3.1.3-5 Simulation Results Comparison on Sensitivity of Processes on Activity

### 3.2 MULTIPLE DRIVERS - CASE 2

**Q**

Why does the code in figure 3.2-1 provide a multiple driver error, yet separate elements of the record are being updated?

```

architecture Recrd_a of Recrd is
  type Rcd_Typ is record
    A : Integer;
    B : Integer;
  end record;

  type RcdAry_Typ is array(0 to 15) of Rcd_Typ;
  Signal RcdAry_s : RcdAry_Typ;
  signal Count      : Natural;
begin
  Count <= Count mod 16 after 10 ns;

  A_Lbl: process
  begin
    RcdAry_s(Count).A <= 1;           ← source for RcdAry_s
    wait;
  end process A_Lbl;

  B_Lbl: process
  begin
    RcdAry_s(Count).B <= 2;           ← # ERROR: Nonresolved signal
    wait;
  end process B_Lbl;
end Recrd_a;

```

RcdAry\_s already has a source (on line 18).

Figure 3.2-1 Multiple Drivers (drivers\recrd.vhd)

**A**

This case is similar to the one described in section 3.1. The index of the array *RcdAry\_s* is *Count*. That index is not static and is calculated dynamically. The compiler creates a driver for all the indexes of the composite object. In other words, process *A\_Lbl* attaches a driver for all the elements of *RcdAry\_s*. Similarly, process *B\_Lbl* attaches a driver for all the elements of *RcdAry\_s*. There is a conflict caused by multiple drivers on an unresolved object. If the code is changed as shown in figure 3.2-2, then there is no driver conflict:

```

A_Lbl: process
begin
  RcdAry_s(0).A <= 1;
  wait;
end process A_Lbl;

B_Lbl: process
begin
  RcdAry_s(0).B <= 2;
  wait;
end process B_Lbl;

```

Figure 3.2-2 Multiple drivers Resolution with Static Indices  
(drivers\recrdok.vhd)

### 3.3 MULTIPLE DRIVERS ERROR – CASE 3

**Q**

Why does the code fragment in figure 2.3-1 yield a multiple driver problem? It also appears that every element in signal *U* has a separate driver and should be resolved separately. If the line *U(I'right) <= I(I'right);* is commented out, then there is no error.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity Bug is
    port ( I : in     Std_uLogic_Vector(0 to 3) := "0000";
           Y : out    Std_uLogic);
end Bug;

architecture Structure of Bug is
    signal U : Std_uLogic_Vector(I'range);
begin
    U(I'right) <= I(I'right);

    X1_Lbl: for X in I'right - 1 downto I'left generate
        U(X) <= U(X + 1) or I(X);      -- ☺ VHDL'93; -- ⚡* VHDL'87
    end generate X1_Lbl;

    Y <= U(I'left);
end Structure;
```

Figure 3.3-1 Multiple Drivers Error (drivers\bug.vhd)

**A**

In this case, the LRM for VHDL'87 does not agree with the intent of the language. Specifically, the expression *I'right* is not globally static<sup>8</sup> because it is not one of the globally static primaries listed on page 7-16 of IEEE Std. 1076-1987. Although *I'right* is a predefined attribute of an object of a globally static subject, it is not a predefined attribute of a globally static subtype, because the prefix *I* denotes an object, not a subtype. Therefore, the longest static prefix of the target name *U(I'right)* is *U*, not *U(I'right)*. The concurrent signal assignment *U(I'right) <= I(I'right)* creates drivers for all scalar subelements of *U*. When the assignment under the *generate* statement creates drivers, they are duplicate drivers. If the code is elaborated using a VHDL-87 simulator, it is proper for a simulator to report the error.

In VHDL-93 this problem was corrected. Specifically, LRM'93 7.4.2 states that [1] an expression is said to be globally static if it is (among other things) 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.

Not all simulators report the error because some vendors implemented the intent of the language rather than the letter of the law of the language. Such an implementation will yield non-portable code. The use of a single process alleviates this multiple driver problem. This is shown in figure 3.3.-2.

```

architecture Structure of Bug is
  signal U : Std_Logic_Vector(I'range);
begin
  Drive_Lbl: process (U, I) begin
    U(I'right) <= I(I'right);
    for x in I'right-1 downto I'left loop
      U(x) <= U(x+1) or I(x);
    end loop;
  end process Drive_Lbl;

  Y <= U(I'left);
end Structure;

```

One concurrent statement instead of two

Figure 3.3-2 Fix of Bug (drivers\bugfix.vhd)

### 3.4 MULTIPLE DRIVERS ERROR -- COMPONENT

**Q**

The multiple component instantiation (shown in figure 3.4-1) causes a multiple driver error, although the individual bits are not driven in the architecture. Where is the error?

```

entity Test is
  generic(BitsS : Integer := 1);
  port(A : out Bit_Vector(7 downto 0));
end Test;

architecture Test_a of Test is
  constant K : Integer := BitsS;
begin
  A(K) <= '1';
end Test_a;

entity TestTB is
end TestTB;

architecture TestTB_a of TestTB is
  component Test
    generic(BitsS : Integer := 1);
    port(A : out Bit_Vector(7 downto 0));
  end component;

  signal A : Bit_Vector(7 downto 0);
begin
  U1_Test: Test
    generic map(BitsS => 1)
    port map(A => A);

  U2_Test: Test
    generic map(BitsS => 2)
    port map(A => A);
  -- Multiple Drivers Error
end TestTB_a;

```

A(1) driven in architecture

A(2) driven in architecture

Figure 3.4-1 Multiple Drivers (driver\test.vhd)

**A**

An *OUT*, *INOUT*, or *BUFFER* port is a **SOURCE** of data regardless of the architecture of the component. Thus, even for an empty architecture, there is a source or driver for each element of the array. This concept is shown in figure 3.4-2.

```

Begin
  U1_Test: Test
    generic map(Bits => 1)
    port map(A => A);
  ← Port A is a SOURCE on all elements

  U2_Test: Test
    generic map(Bits => 2)
    port map(A => A);
  ← Port A is a SOURCE on all elements
    -- Multiple Drivers Error ⚡
end TestTB_a;

```

**Figure 3.4-2 Sources of Multiple Drivers**

Figure 3.4-3 provides a solution to this problem by using the resolved *Std\_Logic\_Vector* type, and by driving all bits to 'Z' from within the architecture.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity Test is
  generic(Bit_g : Integer := 1);
  port(A : out Std_Logic_Vector(7 downto 0));
end Test;

architecture Test_a of Test is
begin
  A <= (others => 'Z');
  A(Bit_g) <= '1';
end Test_a;

library IEEE;
use IEEE.Std_Logic_1164.all;
entity TestTB is
end TestTB;

architecture TestTB_a of TestTB is
component Test
  generic(Bit_g : Integer := 1);
  port(A : out Std_Logic_Vector(7 downto 0));
end component;

signal A : Std_Logic_Vector(7 downto 0);
begin
  U1_Test: Test
    generic map(Bit_g => 1)
    port map(A => A);

  U2_Test: Test
    generic map(Bit_g => 2)
    port map(A => A);
end TestTB_a;

```

**Figure 3.4-4 Resolution of Multiple Driver Issue (driver\testok.vhd)**

### 3.5 CODING STYLE FOR DETECTION OF MULTIPLE DRIVERS

**Q** Since synthesis does not allow multiple drivers within a chip design, what coding style can detect a multiple driver condition?

**A** The use of unresolved types will ensure automatic detection of multiple drivers by a compiler compliant with IEEE-1076 standard. This is because, as per LRM,

[1] if more than one driver exists for a signal, then that signal MUST be a resolved type.

For synthesizable designs, the following unresolved types can be used for all variables, signals and ports:

Type	Package	Comments
Std_uLogic	IEEE.Std_Logic_1164	Std_uLogic is a subtype of Std_Logic.
Std_uLogic_Vector	IEEE.Std_Logic_1164	This type is a different (but closely related) type to Std_Logic_Vector.
Integer	Std.Standard	Subtypes based on integer type can also be used.
Boolean	Std.Standard	
Enumerated	User defined	
Unsigned Signed ? 	User_Pkg.Synopsys	Synopsys package operates on objects of type Bit_Vector. Package may not be supported by synthesis vendors.
Unsigned Signed ? 	IEEE.Numeric_Bit	Operates on objects of type Bit_Vector. From IEEE Standard VHDL Synthesis Package (1076.3, Numeric_Bit). Package may not yet be supported by synthesis vendors.

A component with *out* or *inout* ports of unresolved types cannot be directly interfaced to signals at a higher level of hierarchy when other drivers are associated with those signals. However, type conversions in port associations (see section 1.4.4) provide a mechanism to convert between resolved and unresolved types. Figure 3.5-1 demonstrates the concept of instantiating a component with unresolved ports in an environment with resolved signals.

There is a general problem in using *Std\_uLogic* and *Std\_uLogic\_Vector* with arithmetic operations. This is because the standard IEEE (and the Synopsys-supplied) packages do not fully overload all the functions for both flavors of (resolved and unresolved) standard logic. Specifically, only types *Std\_Logic* and *Std\_Logic\_Vector* are fully implemented, while *Std\_uLogic* and *Std\_uLogic\_Vector* are not.

The designer must either use the resolved standard logic type throughout the design, or resort to conversion functions to translate between unresolved types and resolved types. It is important to note that not all synthesis vendor fully support those conversion functions. Figure 3.5-2 demonstrates the levels of complexity. Because of these difficulties in

synthesis, coding style, and the many levels of type conversions, many users have abandoned the use of the unresolved standard logic, particularly when modeling designs that involved arithmetic operations.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

entity Uresolve is
  port(A : in    Std_Logic_Vector(2 downto 0);
       B : in    Std_Logic;
       C : out   Std_Logic;
       D : out   Std_Logic_Vector(2 downto 0)
      );
end Uresolve;

architecture Uresolve_a of Uresolve is

  function AND_Reduce(ARG: Std_Logic_Vector) return Std_Logic is
    variable Result_v: Std_Logic;
  begin
    Result_v := '1';
    for I in ARG'range loop
      Result_v := result_v and ARG(I);
    end loop;

    return Result_v;
  end;

  signal A_s : Std_Logic_Vector(2 downto 0);

begin  -- Uresolve_a
  A_s <= not A;
  C <= AND_Reduce(A_s);
  D <=     A_s when B = '1'
        else (others => 'Z');
end Uresolve_a;

-----
-- Testbench VHDL'93
-----
library IEEE;
use IEEE.Std_Logic_1164.all;

entity TESTBENCH is
begin
end TESTBENCH;

architecture TESTBENCH_A of TESTBENCH is
  component Uresolve
    port(
      A : in    Std_Logic_Vector(2 downto 0);
      B : in    Std_Logic;
      C : out   Std_Logic;
      D : out   Std_Logic_Vector(2 downto 0)
     );
  end component;

  signal A : Std_Logic_Vector(2 downto 0);
  signal B : Std_Logic := '0';
  signal C : Std_Logic;
  signal D : Std_Logic_Vector(2 downto 0);

```

← Unresolved types

```

begin
  Uresolve_I: Uresolve
  port map (
    A          => Std_uLogic_Vector(A),
    B          => B,
    Std_Logic(C)  => C,
    Std_Logic_Vector(D) => D);

  A <= "101" after 10 ns,
  "011" after 20 ns,
  "000" after 35 ns,
  "100" after 50 ns;

  B <= not B after 5 ns;

  C <= 'Z' after 2 ns,
  '1' after 7 ns,
  'Z' after 11 ns,
  '0' after 17 ns;

  D <= "100" after 4 ns,
  "ZZZ" after 8 ns,
  "000" after 22 ns,
  "111" after 30 ns,
  "ZZZ" after 40 ns;
end TESTBENCH_A;

```

Type conversion  
(VHDL'93) in port

**Figure 3.5-1 Instantiating a Component with Unresolved Ports  
(Uresolve.vhd)**

```

library Work;
use Work.Synopsys.all;

library IEEE;
use IEEE.Std_Logic_1164.all;

entity Uresolv2 is
  port(A : in      Std_uLogic_Vector(7 downto 0);
       B : inout   Std_uLogic_Vector(7 downto 0);
       Enb : in     Std_uLogic;
       C : out      Std_uLogic_Vector(7 downto 0)
     );
end Uresolv2;

architecture Uresolv2_a of Uresolv2 is
  signal A_s : Signed(7 downto 0); -- array of Bit
  signal B_s : Signed(7 downto 0);
  signal C_s : Signed(7 downto 0);
begin -- Uresolv2_a
  A_s <= Signed(To_BitVector(A));
  B_s <= Signed(To_BitVector(B));
  C_s <= A_s + B_s;

  C <= To_StdULogicVector(Bit_Vector(C_s)) when Enb = '1'
    else (others => 'Z');
end Uresolv2_a;

```

- Complex type translations
- Synthesis is not guarantee because of lack of package support

**Figure 3.5-2 Levels of Complexity when using Unresolved signal and Arithmetic Operations (Uresolv2.vhd)**

## **4. SUBPROGRAMS**

---

---

This section discusses issues related to side effects, memory leaks, types and file declarations in subprograms, conversion function, and variable normalization.

## 4.1 SIDE EFFECTS FROM A PROCEDURE

**Q**

There is strong desire to maintain the procedure argument lists to a minimum. However, from within a procedure there is a need to read signals and assign values to signals not declared in the procedure parameters. The use of side effects significantly reduces the procedure interface list during the procedure calls.

To enhance reusability, it would be desired to declare those procedures (with the side effects) in packages, or in the architecture declarative block. However, the following error message is displayed at compilation: "*Cannot drive signal from this subprogram.*"

1. Which signals can be used as side effects?
2. What are the issues?
3. Are there better alternate solutions?

**A**

There are two aspects to the issue of side effects: The language and the style. These views are examined in the following subsections.

### 4.1.1 Language View

Procedures with side effects are defined as procedures that can read and modify variables and signals not declared in the formal parameter list. To answer the questions on the use of side effects, it is important to understand the scope of visibility rules. These are demonstrated in figure 4.1.1-1. All procedures can have READ (i.e., NO assignment) side effects on signals and variables that are within the visibility rules of the procedure. This would include:

1. Global signals when the procedure is defined in a package.
2. Signals visible by the architecture (ports, signals of the architecture, global signals) when the procedure is declared in the declarative part of the architecture.
3. Signals and local variables visible by a process when the procedure is declared in the declarative part of a process.

Procedures declared in packages and in the declarative part of an architecture cannot have side effects with signal assignments; this is because the drivers required for the assignment do not belong to any process. A procedure may be called by several processes. LRM 8.4.1 states: [1] *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 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.*

A procedure may have READ and WRITE side effects with signal and variable

assignments if that procedure is declared in the declarative part of a process. A procedure has visibility over the process variables, and any signal assignment made within that procedure has its DRIVER associated with that process.

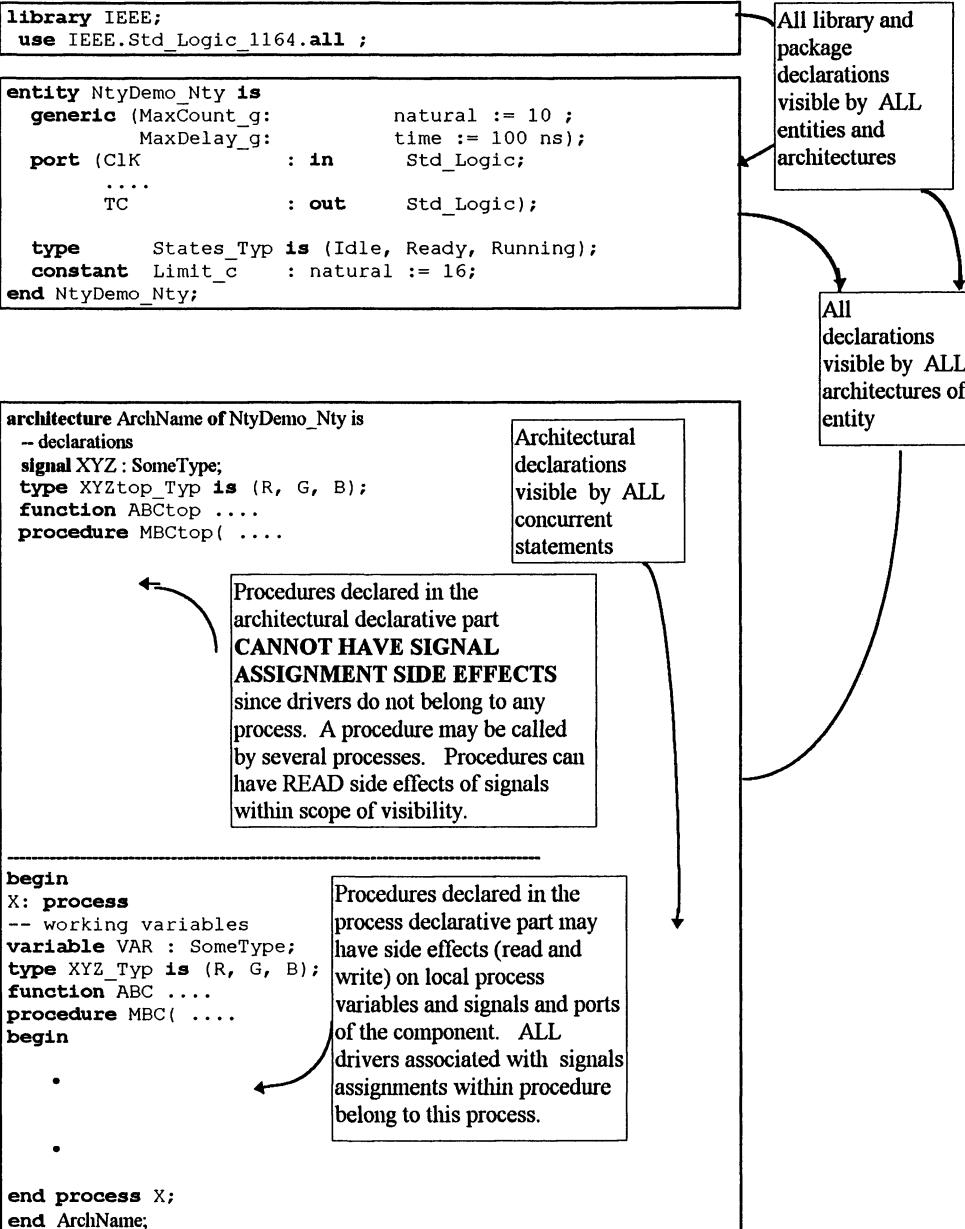


Figure 4.1.1-1 Scope of Visibility Rules.

Note that when a procedure has a formal parameter of class SIGNAL, and the direction of that formal parameter is **out** or **inout**, then an association is made between the actual parameter and the formal parameter. If that procedure is called from with a process, then the process owns the driver for the associated signal (the actual). If that procedure is used as a concurrent procedure, that procedure has an equivalent "process," and the driver for the associated signal (the actual) is owned by that equivalent process.

Figure 4.1.1-2 demonstrates an example of a procedure declared in the process declarative region of a process. Figure 4.1.1-3 is a graphical representation of this component with its side effects.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity SideEffect is
  port (Data      : inout  Std_Logic_Vector(7 downto 0);
        Address   : out    Std_Logic_Vector(7 downto 0);
        RdF       : out    Std_Logic;
        DataRdy   : out    Std_Logic
      );
end SideEffect;

architecture SideEffect_a of SideEffect is
  signal Clk  : Std_Logic := '0';
begin
  Job_Lbl: process
    procedure GetData    -- WITH SIDE EFFECTS
      (constant Address_c : in Std_Logic_Vector(7 downto 0)) is
    begin
      Data      <= (others => 'Z');
      Address  <= Address_c;
      RdF      <= '1';
      DataRdy  <= '1';

      wait until Clk'event and Clk = '1';
      Data      <= (others => 'Z');
      Address  <= (others => 'H');
      RdF      <= '1';
      DataRdy  <= '0';
    end GetData;

    begin
      wait until Clk'event and Clk = '1';
      GetData(Address_c => "00001010");
      wait;
    end process Job_Lbl;

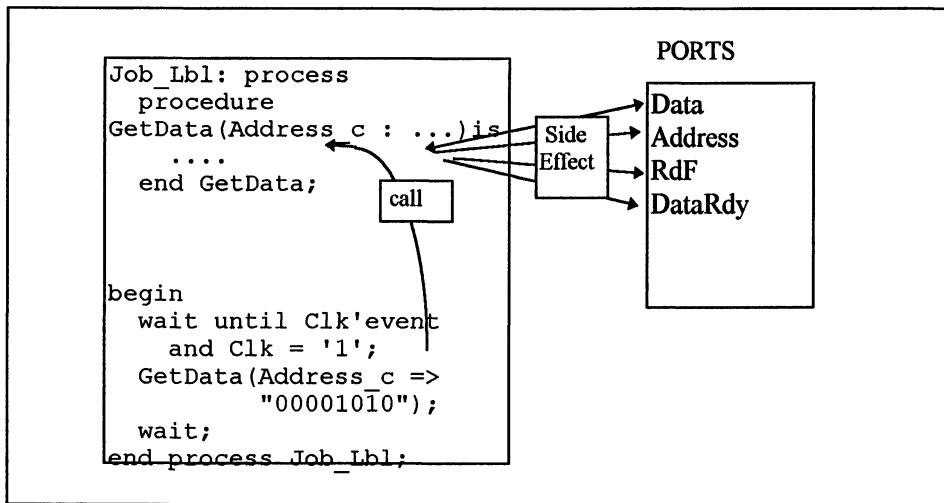
  Clk <= not Clk after 50 ns;
end SideEffect_a;

```

 Procedure has side effects because signal assignments are made to signals not declared in the formal parameter list. Those signals are visible by the process. All drivers to those signals belong to this process.

 It is poor style to use constrained array in formal parameters of subprograms

Figure 4.1.1-2 Architecture with Side Effects (subprgm\sidefct.vhd)



**Figure 4.1.1-3 Graphical Representation of Component with its Side Effects**

#### 4.1.2 STYLE VIEW

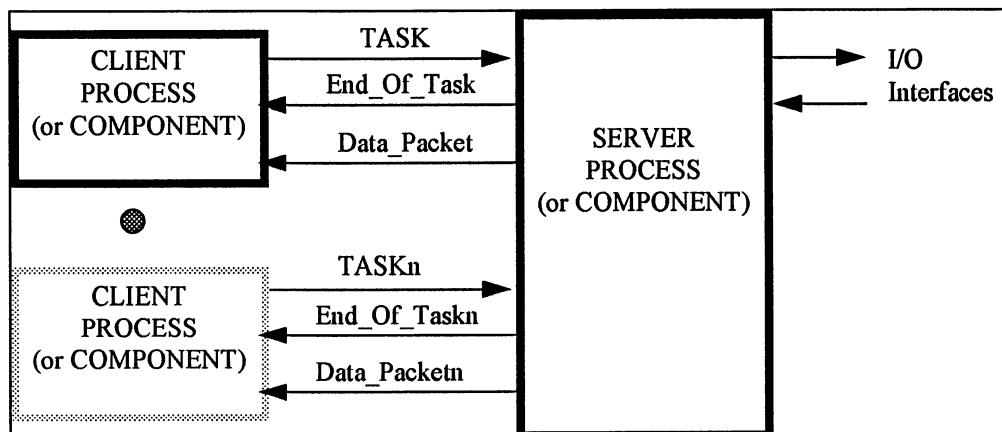
**M** In general, it is best to avoid procedures with side effects. Such procedures are less readable, maintainable and reusable, and may be error prone because the user may not be aware of ALL of the side effects. One method to avoid side effects (but yet maintain a concise procedure interface) is to use an object oriented solution with a client and server type of interface mechanism (shown in figure 4.1.2-1). Instead of a procedure, one could use either a process or a component that acts as a server for the handling of low level interfaces. This is directed by clients through task requests. The server has visibility (read and write) to all signals accessible within its architecture. If the server is a process, then all the architectural signals, ports, and visible global signals are accessible. If the server is a component, then the server has access to all the signals connected to its ports, and to any global signal made visible to the component.

The client process or component is responsible for the definition of the high level tasks, such as the read request at a specific address, or the writing of  $n$  words of data to a specific location. The client passes the task request to the server who is then responsible to prioritize the tasks (if more than one client needs a task performed), and to twiddle the interface signals and ports to honor the high level request. This low level interface can be fairly complex, such as the control of the *Read/Write* signals, bus *request/grant*, block formatting, parity generation/checking, etc. The server alerts the client of its completion through an *end\_of\_task* handshaking signal. Any data returned to the server is formatted and transferred to the client through the *data\_packet* signal. It is recommended that both, the client tasks and server data packets, be of a record type because the information is more concisely and logically handled.

The client/server is a familiar concept experienced at a restaurant. The customer (or client) makes a food order request (the TASK) such as ordering a hamburger, well done, with pickles and tomatoes. This order form (the record) is then passed to the cook (the server) who is responsible for placing the uncooked meat on the hot grill, waiting until one side cooks, turning it over, waiting until the other side cooks, cutting the bread bun, ...., and finally putting the desired meal on the tray (the I/O Interfaces), and alerting the customer (the End Of Task) that the order is done.

This client/server concept is very useful for the modeling of non-synthesizable bus functional models (BFM) where there is a clear distinction between a definition of the desired high level transactions (the client) and the low level interfaces (the server). This concept of client/server is expanded in reference [9]<sup>9</sup> for the definition of BFM and testbenches. Chapter 10 of that book describes the methodology and style on this issue, while chapter 11 provides a complete example using a Universal Asynchronous Receiver Transmitter (UART) model in a testbench environment. Section 8.5 further elaborates the design of a BFM using this concept.

A client/server model is shown in figure 4.1.2-2, and graphically in figure 4.1.2-3.



**Figure 4.1.2-1 Client/Server Type of Interface as an Alternative to Side Effects**

```

library IEEE;
use IEEE.Std_Loic_1164.all;
entity Hide is
    port(Data : inout Std_Loic_Vector(7 downto 0);
         Address : out Std_Loic_Vector(7 downto 0);
         RdF : out Std_Loic;
         DataRdy : out Std_Loic
    );
end Hide;

```

```

architecture Hide_a of Hide is
  type Action_t typ is (Read, Write);
  type Task_Typ is record
    Address : Std_Logic_Vector(7 downto 0);
    Data    : Std_Logic_Vector(7 downto 0);
    Action   : Action_t typ;
  end record;

  signal Task_s : Task_Typ;
  signal Clk     : Std_Logic := '0';

begin -- Hide_a
  -- Purpose: Accepts a high level task and converts it
  --           : to low level control signals
  -----
  Task_Lbl : process
  begin -- process Task_Lbl
    wait on Task_s'transaction;
    case Task_s.Action is
      when Read =>
        Data    <= (others => 'Z');
        Address <= Task_s.Address;
        RdF     <= '1';
        DataRdy <= '1';
        wait until Clk'event and Clk = '1';
        Data    <= (others => 'Z');
        Address <= (others => 'H');
        RdF     <= '1';
        DataRdy <= '0';

      when Write =>
        Data    <= Task_s.Data;
        Address <= Task_s.Address;
        RdF     <= '0';
        DataRdy <= '1';
        wait until Clk'event and Clk = '1';
        Data    <= (others => 'Z') ;
        Address <= (others => 'H');
        RdF     <= '1';
        DataRdy <= '0';
    end case; -- Task_s.Action
  end process Task_Lbl;
  -- Purpose: Provides high level jobs to be executed on the bus
  HighLevelJob_Lbl : process
    variable Task_v : Task_Typ;
  begin -- process HighLevelJob_Lbl
    wait until Clk'event and Clk = '1';
    Task_v.Address := "10100011";
    Task_v.Action  := Read;
    Task_s <= Task_v; -- send task
    wait;
  end process HighLevelJob_Lbl;
  Clk <= not Clk after 50 ns;
end Hide_a;

```

**The SERVER**

**The CLIENT**

Tasks are defined using a variable.  
They are then submitted with a single signal assignment of the whole record. This enhances simulation speed.

**Figure 4.1.2-2 Client/server Approach to Avoid Side Effects  
(subprgm\drvproc.vhd)**

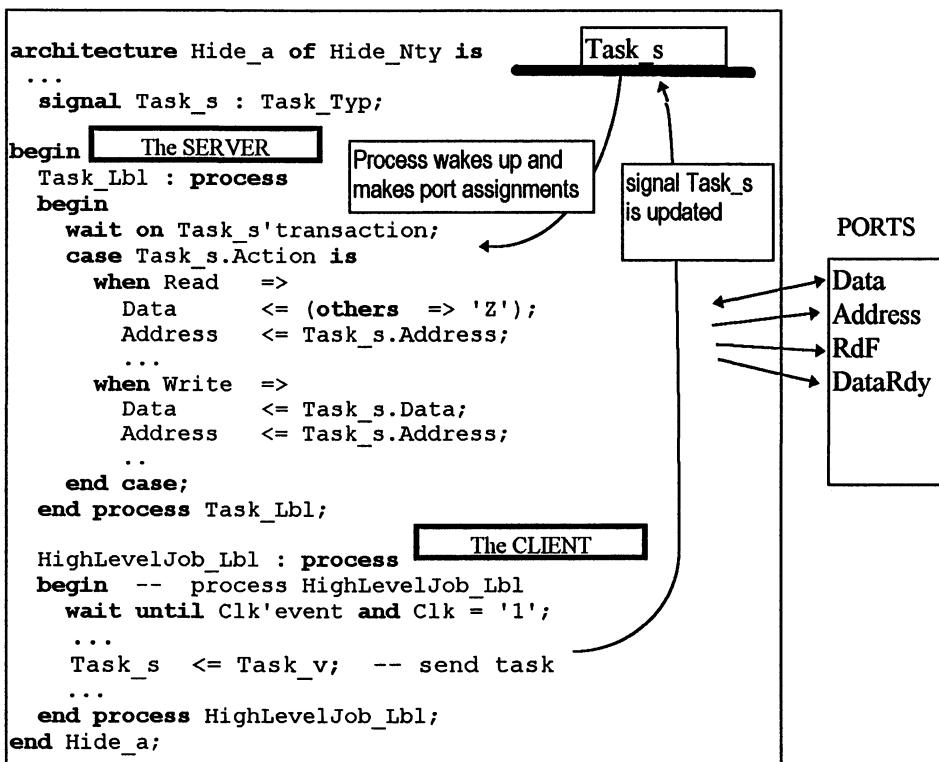


Figure 4.1.2-3 Client/Server Approach to Avoid Side Effects

## 4.2 GARBAGE COLLECTION OF DYNAMICALLY CREATED OBJECTS

**Q**

If a function declares a local variable of an access type, and a new dynamic object is created within that function (with the *new* as shown below), then what happens to the created object if the function returns without a deallocation? Specifically, when the function leaves the scope of the program after the call, are ALL the objects (declared locally in the function) destroyed and deallocated (i.e., automatic garbage collection), or are only the local variables destroyed? Figure 4.2-1 demonstrates an example of dynamically linked object in a subprogram.

```

Function Image(In_Image : Time) return String is
    variable L : Std.TextIO.Line; -- access type
begin
    -- the WRITE procedure creates an object with "NEW".
    -- L is passed as an output of the procedure.
    Std.TextIO.WRITE(L, In_Image);

    -- No deallocation of L (e.g. Deallocate L)
    return L.all;   ↳ Memory LEAK !!!
end Image;

```

Figure 4.2-1 Leaky Dynamically Linked Object in a Subprogram.

**A** The above code fragment will leak because this dynamic memory allocation is not specifically retrieved with a `deallocate` statement. The language does not require a simulator to track dynamically allocated memory. A simulator vendor may choose to either track these memories or to ignore them. This generally does not represent a problem unless the size of the memory allocated is large, or there are several calls to the subprograms. Figure 4.2-2 represents a method to eliminate this memory leak.



It is recommended to maintain garbage collection in subprograms.

*Rationale:* Large memory leaks may cause a simulator to exhaust all the memory resources of the host.

```

entity Image is
end Image;

Architecture Image_a of Image is
  function Image(In_Image : Time) return String is
    use STD.TextIO.all;
    variable L : Line; -- access type
    variable W : String(1 to 25) := (others => ' ');
    -- Long enough to hold a time string
  begin
    -- the WRITE procedure creates an object with "NEW".
    -- L is passed as an output of the procedure.
    Std.TextIO.WRITE(L, In_Image);

    -- Copy L.all on W
    W(L.all'range) := L.all;
    Dynamically allocated  
memory is reclaimed here
    Deallocate(L);
    return W;
  end Image;

begin
  Image_Lbl: process
  begin
    wait for 1.25 us;
    assert False
      report "Time is " & Image(now)
      severity Note;
  end process Image_Lbl;
end Image_a;

```

**Figure 4.2-1 Non-Leaky Dynamically Linked Object in a Subprogram(subprgm\image.vhd)**

Note that the above image function is not needed for VHDL'93 since the attribute `T'Image(X)` converts a scalar object X of type T into a string.

Figure 4.2-2 demonstrates an aggravated case of a memory leak . On a PC with 16 MB of physical memory, using Model Technology simulator, this model experienced a fatal system error after 300 function calls

```

library STD;
use Std.TextIO.all;
use Std.TextIO;
...
architecture Access_Beh of Access_Nty is
    function Test(Count_c : Integer) return Integer is
        variable L_v : TextIO.Line;
    begin
        L_v := new String(1 to 50000); -- Large string allocation
        return Count_c + 1; -- no deallocation of L_v
    end Test;

    signal T_s : Integer := 0;
begin -- Access_Beh
    Demo_lbl : process
    begin -- process Demo_lbl
        T_s <= Test(T_s);
        wait for 1 ns;
    end process Demo_lbl;
end Access_Beh;

```

This function "LEAKS" because the dynamic memory is never recovered with the deallocate statement.

**Figure 4.2-2 Aggravated Case of a Leaky Subprogram (subprgm\Access.vhd)**

If the function is modified to reclaim the dynamically allocated memory, then the model does not exhibit a fatal error as shown in figure 4.2-3

```

function Test(Count_c : Integer) return Integer is
    variable L_v : TextIO.Line;
begin
    L_v := new String(1 to 50000); -- Large string
    Deallocate(L_v); -- Reclaim the dynamic memory
    return Count_c + 1; -- no deallocation of L_v
end Test;

```

**Figure 4.2-3 Non-Leaky Subprogram (subprgm\AccessOK.vhd)**

As a side note, a procedure that has the potential for memory leaks is the *TextIO.ReadLine*. This procedure, as implemented by one vendor, was found to cause serious problems when large files are read because the host eventually runs out of resources, and the simulation stops. The problem is caused by the fact that, in some implementations, the *ReadLine* procedure does not check that the line pointer (being passed to the procedure) may contain data. The procedure just reads a new line and assigns the pointer to that line, leaving the old data dangling in the system, and thus causing the memory leak. The VHDL LRM is non specific about the implementation of the *ReadLine* procedure. Model Technology correctly prevents a memory leak problem because inside the *ReadLine* procedure (implemented in C), the line is first deallocated (memory retrieved back to the system) if it is not empty. Figure 4.2-4 provides a sample code to test for a memory leak condition (file *wrln87.vhd* and *wrln93.vhd*).

```

library Std;
use Std.TextIO.all;
entity FileIO is
end FileIO;

architecture FileIO_a of FileIO is
procedure ReadLn(constant FileIn : in String) is
  file DataIn_f : Text is in FileIn;
  variable Line_v : Line; -- pointer to string
begin
  while not Endfile(DataIn_f) loop
    ReadLine(DataIn_f, Line_v); ←
  end loop;
end ReadLn;

begin
  ReadFile_Lbl : process
    variable I_v : Natural := 0;
    variable L_v : Std.TextIO.Line;
  begin -- process ReadWriteFile_Lbl
    for I in 1 to 50000 loop
      ReadLn(FileIn => String'("datain.txt")); -- User must supply text file
      I_v := I_v + 1;
      Write(L_v, I_v);
      WriteLine(Output, L_v);
    end loop;
    wait;
  end process ReadFile_Lbl;
end FileIO_a;

```

Procedure may leak if vendor does no check that *Line\_v* is null prior to reading a new line.

**Figure 4.2-4 Code to Test for a Memory Leak Condition  
(subprgm\wrln87.vhd)**

To fix the *ReadLine* problem for a product that does leak, use one of the following solutions:

1. Prior to the *ReadLine* call, add the following code:

```

if L /= null then
  deallocate (L);
end if;
ReadLine(F, L);

```

2. Use the following *Readline* procedure instead:

```

procedure ReadLine_NoLeak(variable F: TEXT; --VHDL'87
                           variable L: inout LINE) is
begin
  if L /= null then
    deallocate (L);
  end if;
  READLINE(F, L);
end;

```

File *subprgm\wrifix87.vhd* (on disk) demonstrates a complete application example for this solution.

### 4.3 ACCEPTABLE TYPES IN PARAMETER LISTS FOR FUNCTION CALLS

**Q**

Why does the function shown in figure 4.3-1 fail to compile?

```
function Test(variable L      : Std.TextIO.Line;
              F      : Std.TextIO.Text;
              signal S   : Natural;
              I      : Natural) return Integer is
begin
  if S'transaction then
    return L.all'Length;
  elsif L.all'length <= I then
    return Character'pos(L.all(I));
  else
    return - 1;
  end if;
end Test;
```

**Figure 4.3-1 Function with Compilation Errors**

**A**

```
function Test(variable L      : Std.TextIO.Line; -- ⚡
              F      : Std.TextIO.Text; -- ⚡
              signal S   : Natural;
              I      : Natural) return Integer is
begin
  if S'Transaction then -- ⚡
  ...
end Test;
```

This function has several errors. Specifically,

1. Formal parameters of functions may not be of class *variable*. Then can only be of class *constant* or *signal*. Formal parameters of functions can only be of mode *in*. If the class is not explicitly mentioned, the default class is *constant*.
2. In subprograms (i.e., functions and procedures), the class *variable* must be used for formal identifiers of type *access* (e.g., *Std.TextIO.Line*) because the actual parameter can only be of class *variable*. An object of *access* type cannot be a *signal* or a *constant* or an *expression*.
3. In VHDL'87, the class *variable* must be used for formal identifiers of type *Std.TextIO.Text* because a file object is a member of the *variable* class.
4. [1] *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.* The code example makes use of the '*transaction*' implicit signal, and that is an error. In addition, it uses the '*transaction*' incorrectly since this attribute represents an implicit signal of type *Bit*, and is not a *Boolean*.

The simplest solution to this problem is to convert this function into a procedure as shown in figure 4.3-1 for VHDL'87, and in figure 4.3-2 for VHDL'93.

```

package body Test_Pkg is
  Procedure Test(variable L : in      Std.TextIO.Line;
                  variable F : in      Std.TextIO.Text; -- VHDL'87
                  Signal S : in      Natural;
                  constant I : in      Natural;
                  variable R : out     Integer) is
begin
  if S'active then
    R := L.all'Length;
  elsif L.all'length <= I then
    R := Character'pos(L.all(I));
  else
    R := - 1;
  end if;
end Test;
end Test_Pkg;

```

Class variable used for  
 access type – '87 and '93  
 Class variable used for  
 text type – '87 only

Figure 4.3.1 Corrected Subprogram for VHDL'87 (subprgm\proc87.vhd)

```

package body Test_Pkg is
  Procedure Test(variable L : in      Std.TextIO.Line;
                  file      F :      Std.TextIO.Text; -- VHDL'93
                  Signal S : in      Natural;
                  constant I : in      Natural;
                  variable R : out     Integer) is
begin
  if S'active then
    R := L.all'Length;
  elsif L.all'length <= I then
    R := Character'pos(L.all(I));
  else
    R := - 1;
  end if;
end Test;
end Test_Pkg;

```

Class variable used for  
 access type – '87 and '93  
 Class file used for text type  
 – '93 only

Figure 4.3.2 Corrected Subprogram for VHDL'93 (subprgm\proc93.vhd)

#### 4.4 FILES DECLARATIONS IN PROCEDURES

**Q**

What happens to a file if:

1. The file declarations are in a procedure,
2. Read and write file operations are executed within the procedure, and
3. The procedure is called multiple times.

Are the files closed and reopened at every procedure call?

**A**

The files are reread, and rewritten. This occurs because the declarations of a procedure (files, variables) only exist when the procedure is called. Figure 4.4-1

represents a procedure to copy a file in VHDL'87. Figure 4.4-2 represents the same procedure in VHDL'93.

```

entity FileIO is
end FileIO;

```

```

architecture FileIO_a of FileIO is
procedure CopyFile(constant FileIn : in String;           ← File names are
                    constant FileOut : in String) is
use Std.TextIO.all;
file DataIn_f      : Text is in FileIn;
file DataOut_f     : Text is out FileOut;                ← File are declared
variable OutLine_v : Line; -- pointer to string          in the procedure
begin
  while not Endfile(DataIn_f) loop
    -- Read 1 line from the input file
    ReadLine(DataIn_f, OutLine_v);

    -- Write the line to the out file
    WriteLine(DataOut_f, OutLine_v);
  end loop;
end CopyFile;

begin
  ReadWriteFile_Lbl : process
begin -- process ReadWriteFile_Lbl
  CopyFile(FileIn => String'("datain.txt"),
            FileOut => String'("dataout.txt"));

  wait for 10 ns;

  -- Second access to files.
  CopyFile(FileIn => String'("datain.txt"),
            FileOut => String'("dataout.txt"));
  wait;
end process ReadWriteFile_Lbl;
end FileIO_a;

```

Figure 4.4-1 Procedure to Copy a File in VHDL'87 (subprgm\fcopy87.vhd)

```

entity FileIO is
end FileIO;

architecture FileIO_a of FileIO is
procedure CopyFile(constant FileIn : in String;
                    constant FileOut : in String) is
use Std.TextIO.all;
file DataIn_f      : Text open Read_Mode is FileIn;
file DataOut_f     : Text open Write_Mode is FileOut;
variable OutLine_v : Line; -- pointer to string
begin
  while not Endfile(DataIn_f) loop
    -- Read 1 line from the input file
    ReadLine(DataIn_f, OutLine_v);

    -- Write the line to the out file
    WriteLine(DataOut_f, OutLine_v);
  end loop;
end CopyFile;
begin
  ReadWriteFile_Lbl : process
begin -- process ReadWriteFile_Lbl
  CopyFile(FileIn => String'("datain.txt"),
            FileOut => String'("dataout.txt"));

```

```

wait for 10 ns;

-- Second access to files.
CopyFile(FileIn  => String'("datain.txt"),
          FileOut => String'("dataout.txt"));
wait;
end process ReadWriteFile_Lbl;
end FileIO_a;

```

**Figure 4.4-2 Procedure to Copy a File in VHDL'93 (subprgm\fcopy93.vhd)**

#### 4.5 MULTIPLE ACCESSES OF SAME FILE

**Q** How can one element of a file, and then its preceding element from the same file be accessed (i.e., accessing the *n*th element in a file, and afterwards an element before)?

**A** VHDL'87 does not have a *FILE\_OPEN* procedure like VHDL'93. Files declared in the declaration section of an architecture are opened at the beginning of the simulation and are closed at the end of a simulation. However, if a file is declared in a procedure, then the file is opened every time the procedure is elaborated (or called), and closed at the end of the called procedure. Figure 4.5 demonstrates the concept.

```

library STD;
use STD.TextIO.all;    use STD.TextIO;

entity FileLine is
end FileLine;

architecture FileLine_a of FileLine is
procedure ReadLine(constant FileIn_c : in String;
                    constant LineNumb_c : in Natural;
                    variable OutLine_v : out TextIO.Line;
                    variable Success_v : out Boolean) is
file DataIn_f      : Text is in FileIn_c;
variable Count_v   : Natural;
begin
  Success_v := False;
  while not Endfile(DataIn_f) loop
    -- Read 1 line from the input file
    ReadLine(DataIn_f, OutLine_v);
    Count_v := Count_v + 1;
    if Count_v = LineNumb_c then -- found right line
      Success_v := True;
      exit;
    end if;
  end loop;
end ReadLine;

begin
  ReadFile_Lbl : process
    variable MyLine_v : TextIO.Line;
    variable Success_v : Boolean;
    variable LineNumb_v : Positive;
    constant FileName_c : String := "File1.txt";

```

Annotations:

- Name of a file is passed as a string
- File is declared locally in procedure declarative part
- Lines are read and skipped, until the desired line is reached.

```

begin -- process ReadFile_Lbl
  LineNumb_v := 4;
  ReadLine(FileIn_c => FileName_c,
            LineNumb_c => LineNumb_v,
            OutLine_v => MyLine_v,
            Success_v => Success_v);
  if Success_v then
    WriteLine(Output, MyLine_v);
  end if;

  -- Second attempt, but reading preceding line
  LineNumb_v := LineNumb_v - 1;
  ReadLine(FileIn_c => FileName_c,
            LineNumb_c => LineNumb_v,
            OutLine_v => MyLine_v,
            Success_v => Success_v);
  if Success_v then
    WriteLine(Output, MyLine_v);
  end if;

  wait;
end process ReadFile_Lbl;
end FileLine_a;

```

**Figure 4.5 Multiple Access of a file in VHDL'87 (subprgm\file87.vhd)**

In VHDL'93, it is possible to use the procedures *File\_open*/*File\_close*, thus allowing a file to be explicitly closed and re-opened.

Another solution is to store the previous read data from the file into an array or a linked list (see section 6.1.5 for an example of using linked lists).

## 4.6 FILE ARRAY

**Q**

Is there any way to make an array of file pointers? Ideally, the following is desired:

```

process (S)
  file Cmdfile      : Text is in Cmd_File;
  -- somehow declare Cmdfile as an array of FILEs
begin
  for I in 0 to 15 loop
    ReadLine(Cmdfile(I),Line_in); -- Line_in is a type of LINE
  end loop;
end process;

```

**A**

It is not possible to declare an array of file pointers. The syntax of the file declaration is as follows:

```
-- VHDL'87
file_declaration ::= 
    file identifier : subtype_indication is [ mode ]
        file_logical_name ;
-- VHDL'93
file_declaration ::= 
    file identifier_list : subtype_indication
        file_open_information ;
```

Identifier cannot be an array

The following method is possible:

1. Declare an array of constrained strings,
2. Pass to the procedure the physical file name,
3. Have that procedure declare the file locally,
4. Have the procedure access the locally declared file.

Figure 4.6-1 demonstrates this concept. The input file is reopened from the start at every procedure call. Thus, the line number needs to be passed to the procedure. The procedure must then iterate through all the lines, until it gets to the desired line. Figure 4.6-2 represents the simulation results.

```
library STD;
use STD.TextIO.all;  use STD.TextIO;
--- . . . Empty entity
architecture FileArray_a of FileArray is
procedure ReadLine(constant FileIn_c : in String;
                    constant LineNumb_c : in Natural;
                    variable OutLine_v : out TextIO.Line;
                    variable Success_v : out Boolean) is
    file DataIn_f : Text is in FileIn_c;
    variable Count_v : Natural;
begin
    Success_v := False;
    while not Endfile(DataIn_f) loop
        -- Read 1 line from the input file
        ReadLine(DataIn_f, OutLine_v);
        Count_v := Count_v + 1;
        if Count_v = LineNumb_c then -- found right line
            Success_v := True;
            exit;
        end if;
    end loop;
end ReadLine;

type FileNames_Typ is array(1 to 3) of string(1 to 9);
constant FileNames_c : FileNames_Typ :=
    (1 => "File1.txt",
     2 => "File2.txt",
     3 => "File3.txt");
begin
    ReadWriteFile_Lbl : process
        variable MyLine_v : TextIO.Line;
        variable Success_v : Boolean;
        file DataOut_f : Text is out "FileOut.txt";
```

```

begin -- process ReadWriteFile_Lbl
  for Line_i in 1 to 5 loop
    for File_i in 1 to 3 loop
      ReadLine(FileIn_c => FileNames_c(File_i),
                LineNumb_c => Line_i,
                OutLine_v => MyLine_v,
                Success_v => Success_v);
      if Success_v then
        WriteLine(DataOut_f, MyLine_v);
      end if;
    end loop;
  end loop;
  wait;
end process ReadWriteFile_Lbl;
end FileArray_a;

```

**Figure 4.6-1 Indexed Array Model (subprgm\f\_arr87.vhd)**

file1.txt	file2.txt	file3.txt	fileout.txt
Line 1, file 1	Line 1, file 2	Line 1, file 3	Line 1, file 1
Line 2, file 1	Line 2, file 2	Line 2, file 3	Line 1, file 2
Line 3, file 1	Line 3, file 2	Line 3, file 3	Line 1, file 3
Line 4, file 1	Line 4, file 2	Line 4, file 3	Line 2, file 1
Line 5, file 1	Line 5, file 2	Line 5, file 3	Line 2, file 2
			Line 2, file 3
			Line 3, file 1
			Line 3, file 2
			Line 3, file 3
			Line 4, file 1
			Line 4, file 2
			Line 4, file 3
			Line 5, file 1
			Line 5, file 2
			Line 5, file 3

**Figure 4.6-2 Simulation Results for Array Model**

#### 4.7 CONVERSION FUNCTION FROM INTEGER TO TIME

**Q** How can one write a conversion function from type integer to type time? Conversely, how can one write a conversion function from type time to type integer?

**A** The package shown in figure 4.7 provides these conversion functions. However, it is easier (and recommended) not to call the functions, but apply the conversion algorithm in line with the code. To convert an *integer* to type *time*, a multiplication of the *integer* object by a scale factor is necessary (e.g., *S* <= '1' after *IntVariable* \* 1 ns;). Conversely, to convert an object of type *time* to an *integer*, it is necessary to divide this object by a scale factor (e.g., *IntVariable* := *now* / 1 ns;).

Another potential alternative is the use of the attributes '*val*' and '*pos*'. Note that lowest unit scale for the *Time* definition is 1 *fs*. Thus, *Time*'*val*(1) is 1 *fs*. If an *Integer* object needs to be converted to a type *time* in units of nanoseconds, then it becomes necessary to multiply the *Integer* object by 1E6 (e.g., *wait for Time*'*val*(*IntVariable* \* 1E6);). Conversely, *Time*'*pos*(*now*) yields an *Integer* value of the time in *fs*. A division by 1E6 is necessary to represent this *Integer* value in *ns* (e.g., *IntVariable* = *Time*'*pos*(*now*) / 1E6).

```

package Conversions is
    function cvTime(Int : Integer;
                    ScaleFactor : Time := ns) return Time;
    function cvInt(Tim : Time;
                    ScaleFactor : Time := ns) return Integer;
end Conversions;

package body Conversions is
    function cvTime(Int : Integer;
                    ScaleFactor : Time := ns) return Time is
        begin
            return Int * ScaleFactor;
        end cvTime;

    function cvInt(Tim : Time;
                    ScaleFactor : Time := ns) return Integer is
        begin
            return Tim / ScaleFactor;
        end cvInt;
end Conversions;

library Work;
use Work.Conversions.all;
entity TestTime is
end TestTime;

architecture TestTime_a of TestTime is
    signal Bit_s : Bit;
begin
    Test_Lbl: process
        variable I_v : Natural := 10;
    begin
        Bit_s <= not Bit_s after (I_v/5) * 1 ns; -- Alternative 1
        wait for I_v * 1 ns;
        I_v := (now / 1 ns) + 15; -- Alternative 2
        wait for Time'val(I_v * 1E6); -- Alternative 2
        I_v := Time'pos(now) / 1E6;
        Bit_s <= not Bit_s after Time'val(I_v * 1E6 / 5);

        wait for cvTime(I_v); -- Alternative 3
        I_v := cvInt(now);
    end process Test_Lbl;
end TestTime_a;

```

**Figure 4.7 Conversion: Time to Integer (subprgm\timecnvt.vhd)**

## 4.8 NORMALIZATION IN SUBPROGRAMS

**Q**

What is normalization in subprograms? Why is it important?

**A**

**M** It is a good practice to define formal parameters of subprograms that are arrays as unconstrained (versus constrained) arrays. This yields efficient reuse of the subprogram with different sizes of arrays for the actual parameters. When a subprogram call is made, the direction and range of the actual parameter are passed to the formal parameter. The function must not rely on the direction and range of the actual parameter. An error may occur if the direction and/or range of the actual parameter do not match the assumed range and direction of the formal parameter. Figure 4.8-1 demonstrates this concept. A runtime error occurs because the zero index does not exist in the actual parameter. Normalization provides a method to define the range and direction of local variables based on the length of the actual parameters. This is demonstrated in figure 4.8-2.

```

architecture BitReversal_NG of BitReversal is
    function BitReverse(A : Bit_Vector) return Bit_Vector is
        variable A_v : Bit_Vector(A'range);
        variable B_v : Bit_Vector(A'range);
    begin
        A_v := A;
        B_v := A;
        Lp_Lbl : for K in A'length - 1 downto 0 loop
            A_v(K) := B_v(A'length - 1 - K);
        end loop Lp_Lbl;

        return A_v;
    end BitReverse;

    signal T6d_s : Bit_Vector(7 downto 2) := "100111";
    signal T8u_s : Bit_Vector(1 to 8) := "01001111";
    signal T6u_s : Bit_Vector(1 to 6);
    signal T8d_s : Bit_Vector(7 downto 0);
begin
    -- BitReversal_a
    T6u_s <= BitReverse(T6d_s); -- ⚡*
    T8d_s <= BitReverse(T8u_s); -- ⚡*
end BitReversal_NG;

```

Range is inherited from actual parameter  
and may not be what is assumed

**Figure 4.8-1 Non-Normalized Variables in Subprogram  
(subprm\bitrfnct.vhd)**

```

architecture BitReversal_a of BitReversal is
  function BitReverse(A : Bit_Vector) return Bit_Vector is
    variable A_v : Bit_Vector(A'length - 1 downto 0); -- ⚡⚡
    variable B_v : Bit_Vector(A'length - 1 downto 0); -- ⚡⚡
begin
  A_v := A;
  B_v := A;

  Lp_Lbl : for K in A'length - 1 downto 0 loop
    A_v(K) := B_v(A'length - 1 - K);
  end loop Lp_Lbl;
  return A_v;
end BitReverse;
signal T6d_s : Bit_Vector(7 downto 2) := "100111";
signal T8u_s : Bit_Vector(1 to 8) := "01001111";
signal T6u_s : Bit_Vector(1 to 6);
signal T8d_s : Bit_Vector(7 downto 0);

begin -- BitReversal_a
  T6u_s <= BitReverse(T6d_s); -- ☺
  T8d_s <= BitReverse(T8u_s); -- ☺
end BitReversal_a;

```

It's very IMPORTANT to normalize the range of the loop to control the range and direction of the subprogram local variables.

*Rationale:* The subprogram may lead to errors if the range of the actual parameter is ascending and no normalization is performed.

Range and direction is defined by code.  
Length is inherited from actual parameter

Figure 4.8-2 Normalized variables in Subprogram (subprm\bitrfnct.vhd)

Another technique that is often used to normalize objects is aliasing. For example, in package *Std\_Logic\_1164*, the function *and* has the formal parameters *L*, and *R* normalized with aliases.

```

function "and" ( L,R : Std_Logic_Vector ) return Std_Logic_Vector is
  alias Lv : Std_Logic_Vector ( 1 to L'length ) is L;
  alias Rv : Std_Logic_Vector ( 1 to R'length ) is R;

```

Care should be taken when using aliases since they are generally not supported by synthesis (except for those defined in package *Std\_Logic\_1164* or any other standard package that is supported by the synthesizer tool). An alias [1] represents an alternate name for a named entity. Any assignment to the alias is equivalent to an assignment to the named entity. This concept is demonstrated in figure 4.8-3. However, variables should be used in synthesis for user defined subprograms, or when local computations are necessary in the subprogram.

```
architecture Tx_a of Tx is
  procedure X( variable L : inout Std_Logic_Vector ) is
    alias La : Std_Logic_Vector (L'length downto 1) is L;
  begin
    La := not La; ← An assignment on alias La is equivalent to an assignment on L
  end X;

begin
  process
    variable A : Std_Logic_Vector(1 to 4) := "1011";
    variable B : Std_Logic_Vector(4 downto 1) := "0000";
  begin
    X(A);
    X(B);
    wait;
  end process;
end Tx_a;
```

Figure 4.8-3 Normalization Using Aliases (subprgm\nrmalias.vhd)

## **5. PACKAGES**

---

---

This section provides an image package used to convert objects of type *Bit*, *Bit\_Vector*, *Std\_uLogic*, *Std\_Logic\_Vector*, *Signed*, *Unsigned*, *Integer*, *Real*, and *Time* to strings in binary, integer and hexadecimal formats. This conversion is necessary when displaying values on the screen or in files. It also includes the Multiple Input Signature Register (MISR) package, and a Linear Feedback Shift Register (LFSR) package. An integer random number generation package is also included. The MISR, LFSR, and random number packages are very useful in auto-regression tests as discussed in section eight.

## 5.1 CONVERTING TYPED OBJECTS TO STRINGS

**Q** | How can a typed object of type *Bit*, *Bit\_Vector*, *Std\_uLogic*, *Std\_Logic\_Vector*, *Signed*, *Unsigned*, *Integer*, *Real*, or *Time* be converted to a string?

**A** | In VHDL'93, the '*Image*' attribute provides this conversion for scalar prefixes (e.g., *Bit*, *Std\_Logic*, *Integer*, but NOT for *Bit\_Vector*, *Std\_Logic\_Vector*, *Signed*, and *Unsigned*). A conversion function is required for the array types. In VHDL'87, a conversion function is required for any type. Section 4.2 provides an example of an *image* function to demonstrate the garbage collection of dynamically created objects in subprograms. Figure 5.1 provides a package for the overloaded *image* and *HexImage* functions. The function is called *image* because of the close connotation to the attribute '*image*'. The *HexImage function* converts bit strings to a hexadecimal string. The *IntImage function* converts bit strings to an integer string.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_TextIO.all;
use IEEE.Std_Logic_Arith.all;

library Std;
use STD.TextIO.all;

package Image_Pkg is
    function Image(In_Image : Time) return String;
    function Image(In_Image : Bit) return String;
    function Image(In_Image : Bit_Vector) return String;
    function Image(In_Image : Integer) return String;
    function Image(In_Image : Real) return String;
    function Image(In_Image : Std_uLogic) return String;
    function Image(In_Image : Std_uLogic_Vector) return String;
    function Image(In_Image : Std_Logic_Vector) return String;
    function Image(In_Image : Signed) return String;
    function Image(In_Image : Unsigned) return String;

    function HexImage(InStrg   : String) return String;
    function HexImage(In_Image : Bit_Vector) return String;
    function HexImage(In_Image : Std_uLogic_Vector) return String;
    function HexImage(In_Image : Std_Logic_Vector) return String;
    function HexImage(In_Image : Signed) return String;
    function HexImage(In_Image : Unsigned) return String;

    function DecImage(In_Image : Bit_Vector) return String;
    function DecImage(In_Image : Std_uLogic_Vector) return String;
    function DecImage(In_Image : Std_Logic_Vector) return String;
    function DecImage(In_Image : Signed) return String;
    function DecImage(In_Image : Unsigned) return String;

end Image_Pkg;

```

```
package body Image_Pkg is
    function Image(In_Image : Time) return String is
        variable L : Line; -- access type
        variable W : String(1 to 25) := (others => ' ');
        -- Long enough to hold a time string
    begin
        -- the WRITE procedure creates an object with "NEW".
        -- L is passed as an output of the procedure.
        Std.TextIO.WRITE(L, in_image);
        -- Copy L.all onto W
        W(L.all'range) := L.all;
        Deallocate(L);
        return W;
    end Image;

    function Image(In_Image : Bit) return String is
        variable L : Line; -- access type
        variable W : String(1 to 3) := (others => ' ');
    begin
        Std.TextIO.WRITE(L, in_image);
        W(L.all'range) := L.all;
        Deallocate(L);
        return W;
    end Image;

    function Image(In_Image : Bit_Vector) return String is
        variable L : Line; -- access type
        variable W : String(1 to In_Image'length) := (others => ' ');
    begin
        Std.TextIO.WRITE(L, in_image);
        W(L.all'range) := L.all;
        Deallocate(L);
        return W;
    end Image;

    function Image(In_Image : Integer) return String is
        variable L : Line; -- access type
        variable W : String(1 to 32) := (others => ' ');
        -- Long enough to hold a time string
    begin
        Std.TextIO.WRITE(L, in_image);
        W(L.all'range) := L.all;
        Deallocate(L);
        return W;
    end Image;

    function Image(In_Image : Real) return String is
        variable L : Line; -- access type
        variable W : String(1 to 32) := (others => ' ');
        -- Long enough to hold a time string
    begin
        Std.TextIO.WRITE(L, in_image);
        W(L.all'range) := L.all;
        Deallocate(L);
        return W;
    end Image;

    function Image(In_Image : Std_uLogic) return String is
        variable L : Line; -- access type
        variable W : String(1 to 3) := (others => ' ');
    begin
        IEEE.Std_Lock_Textio.WRITE(L, in_image);
        W(L.all'range) := L.all;
        Deallocate(L);
        return W;
    end Image;
```

An implementation must allow a physical type to include the range -2147483647 to 2147483647 (or 10 characters). It is safe to assume that L.all'range will be less than 25 characters.

```

function Image(In_Image : Std_Logic_Vector) return String is
  variable L : Line; -- access type
  variable W : String(1 to In_Image'length) := (others => ' ');
begin
  IEEE.Std_Logic_Textio.WRITE(L, in_image);
  W(L.all'range) := L.all;
  Deallocate(L);
  return W;
end Image;

function Image(In_Image : Std_Logic_Vector) return String is
  variable L : Line; -- access type
  variable W : String(1 to In_Image'length) := (others => ' ');
begin
  IEEE.Std_Logic_TextIO.WRITE(L, In_Image);
  W(L.all'range) := L.all;
  Deallocate(L);
  return W;
end Image;

function Image(In_Image : Signed) return String is
begin
  return Image(Std_Logic_Vector(In_Image));
end Image;

function Image(In_Image : UnSigned) return String is
begin
  return Image(Std_Logic_Vector(In_Image));
end Image;

function HexImage(InStrg : String) return String is
  subtype Int03_Typ is Integer range 0 to 3;
  variable Result : string(1 to ((InStrg'length - 1)/4)+1) :=
    (others => '0'); -- length of result string
  variable StrTo4 : string(1 to Result'length * 4) :=
    (others => '0'); -- length of extended bit vector
  variable MTspace : Int03_Typ; -- Empty space to fill in
  variable Str4 : String(1 to 4);
  variable Group : Natural := 0;
begin
  MTspace := Result'length * 4 - InStrg'length;
  StrTo4(MTspace + 1 to StrTo4'length) := InStrg; -- padded with '0'
  Cnvt_Lbl : for I in Result'range loop
    Group := Group + 4; -- identifies end of bit # in a group of 4
    Str4 := StrTo4(Group - 3 to Group); -- get next 4 characters
    case Str4 is
      when "0000" => Result(I) := '0';
      when "0001" => Result(I) := '1';
      when "0010" => Result(I) := '2';
      when "0011" => Result(I) := '3';
      when "0100" => Result(I) := '4';
      when "0101" => Result(I) := '5';
      when "0110" => Result(I) := '6';
      when "0111" => Result(I) := '7';
      when "1000" => Result(I) := '8';
      when "1001" => Result(I) := '9';
      when "1010" => Result(I) := 'A';
      when "1011" => Result(I) := 'B';
      when "1100" => Result(I) := 'C';
      when "1101" => Result(I) := 'D';
      when "1110" => Result(I) := 'E';
      when "1111" => Result(I) := 'F';
      when others => Result(I) := 'X';
    end case;
  end loop Cnvt_Lbl;
  return Result;
end HexImage;

```

```
function HexImage(In_Image : Bit_Vector) return String is
begin
    return HexImage(Image(In_Image));
end HexImage;

function HexImage(In_Image : Std_uLogic_Vector) return String is
begin
    return HexImage(Image(In_Image));
end HexImage;

function HexImage(In_Image : Std_Logic_Vector) return String is
begin
    return HexImage(Image(In_Image));
end HexImage;

function HexImage(In_Image : Signed) return String is
begin
    return HexImage(Image(In_Image));
end HexImage;

function HexImage(In_Image : Unsigned) return String is
begin
    return HexImage(Image(In_Image));
end HexImage;

function DecImage(In_Image : Bit_Vector) return String is
    variable In_Image_v : Bit_Vector(In_Image'length downto 1) := In_Image;
begin
    if In_Image'length > 31 then
        assert False
            report "Number too large for Integer, clipping to 31 bits"
            severity Warning;
        return Image(Conv_Integer
                    (Unsigned(To_StdLogicVector
                            (In_Image_v(31 downto 1)))));
    else
        return Image(Conv_Integer(Unsigned(To_StdLogicVector(In_Image))));
    end if;
end DecImage;

function DecImage(In_Image : Std_uLogic_Vector) return String is
    variable In_Image_v : Std_uLogic_Vector(In_Image'length downto 1) := In_Image;
begin
    if In_Image'length > 31 then
        assert False
            report "Number too large for Integer, clipping to 31 bits"
            severity Warning;
        return Image(Conv_Integer(Unsigned(In_Image_v(31 downto 1))));
    else
        return Image(Conv_Integer(Unsigned(In_Image)));
    end if;
end DecImage;

function DecImage(In_Image : Std_Logic_Vector) return String is
    variable In_Image_v : Std_Logic_Vector(In_Image'length downto 1) := In_Image;
begin
    if In_Image'length > 31 then
        assert False
            report "Number too large for Integer, clipping to 31 bits"
            severity Warning;
        return Image(Conv_Integer(Unsigned(In_Image_v(31 downto 1))));
    else
        return Image(Conv_Integer(Unsigned(In_Image)));
    end if;
end DecImage;
```

```

function DecImage(In_Image : Signed) return String is
  variable In_Image_v : Signed(In_Image'length downto 1) := In_Image;
begin
  if In_Image'length > 31 then
    assert False
      report "Number too large for Integer, clipping to 31 bits"
      severity Warning;
    return Image(Conv_Integer(In_Image_v(31 downto 1)));
  else
    return Image(Conv_Integer(In_Image));
  end if;
end DecImage;

function DecImage(In_Image : UnSigned) return String is
  variable In_Image_v : UnSigned(In_Image'length downto 1) := In_Image;
begin
  if In_Image'length > 31 then
    assert False
      report "Number too large for Integer, clipping to 31 bits"
      severity Warning;
    return Image(Conv_Integer(In_Image_v(31 downto 1)));
  else
    return Image(Conv_Integer(In_Image));
  end if;
end DecImage;

end Image_Pkg;

```

**Figure 5.1 Image Package (package\image\_pb.vhd)**

## 5.2 PRINTING OBJECTS FROM VHDL

**Q** How can one create an output on the screen when the objects are of type *real*, *integer*, *Bit\_Vector*, or *Std\_Logic\_Vector*? The format of bit vectors must be in binary or in hex notation. The *assert* statement will almost do the job, but how can the value be easily converted to a string?

**A** There are two methods to produce outputs on the screen:

1. The *Write* and *WriteLine* procedures.
2. The *Assert* statement

The *Image* function, described in section 5.1, or the *Image* attribute (VHDL'93) for scalar prefix only can be used to convert the vector to a string. The *Image* cannot be used for composite type as prefixes (e.g., *Bit\_Vector'Image* is illegal). The *TextIO* package in library *STD* and the Synopsys *Std\_Logic\_TextIO* package *Write* and *WriteLine* procedures for various types (see packages for details). These packages are used in the definition of the *image* function. Figure 5.2-1 demonstrates the application of these concepts for VHDL'87. Figure 5.2-2 demonstrates the same concepts for VHDL'93.

```

library IEEE;
use IEEE.Std_logic_1164.all;

library Work;
use Work.Image_Pkg.all;
use Std.TextIO.all;

entity StringOut is
end StringOut;

```

```

architecture StringOut_a of StringOut is
    signal S_Bit      : Bit;
    signal S_BitV     : Bit_Vector(15 downto 0);
    signal S_Int      : Integer;
    signal S_Real     : Real;
    signal S_uStd     : Std_uLogic;
    signal S_Std      : Std_Logic;
    signal S_StdV     : Std_Logic_Vector(33 downto 0);

begin
    Test_Lbl: process
        variable T_v : Time;
        variable L_v : Line;
    begin
        S_Bit    <= '1';
        S_BitV   <= "0000111101011010";
        S_Int    <= 25;
        S_Real   <= 3.1416;
        S_uStd   <= 'H';
        S_Std    <= 'L';
        S_StdV   <= "111010HLXU00001111HHHHXXXXHHHHLLL1";
        wait for 152 ns;
        -- Method 1, writing to OUTPUT
        Std.TextIO.Write(L_v, String'("T = "));
        Std.TextIO.Write(L_v, now);
        Std.TextIO.Write(L_v, String'(" S_Bit = "));
        Std.TextIO.Write(L_v, S_Bit);
        Std.TextIO.WriteLine(Output, L_v);

        Std.TextIO.Write(L_v, String'("T = "));
        Std.TextIO.Write(L_v, now);
        Std.TextIO.Write(L_v, String'(" S_BitV = "));
        Std.TextIO.Write(L_v, S_BitV);
        Std.TextIO.WriteLine(Output, L_v); ←———— Output is a file class defined in
                                         package TextIO. It is defined as the
                                         standard output, usually the screen.

        Std.TextIO.Write(L_v, String'("Thex = "));
        Std.TextIO.Write(L_v, now);
        Std.TextIO.Write(L_v, String'(" S_BitV = ") & HexImage(S_BitV));
        Std.TextIO.WriteLine(Output, L_v);

        Std.TextIO.Write(L_v, String'("T = "));
        Std.TextIO.Write(L_v, now);
        Std.TextIO.Write(L_v, String'(" S_Int = "));
        Std.TextIO.Write(L_v, S_Int);
        Std.TextIO.WriteLine(Output, L_v); ←———— Use of the HexImage
                                         function for HEX display

        Std.TextIO.Write(L_v, String'("T = "));
        Std.TextIO.Write(L_v, now);
        Std.TextIO.Write(L_v, String'(" S_Real = "));
        Std.TextIO.Write(L_v, S_Real);
        Std.TextIO.WriteLine(Output, L_v);

        Std.TextIO.Write(L_v, String'("T = "));
        Std.TextIO.Write(L_v, now);
        Std.TextIO.Write(L_v, String'(" S_uStd = "));
        IEEE.Std_Logic_TextIO.Write(L_v, S_uStd);
        Std.TextIO.WriteLine(Output, L_v);

        Std.TextIO.Write(L_v, String'("T = "));
        Std.TextIO.Write(L_v, now);

```

```

Std.TextIO.Write(L_v, String'(" S_Std = "));
IEEE.Std_Lock_TextIO.Write(L_v, S_Std);
Std.TextIO.WriteLine(Output, L_v);

Std.TextIO.Write(L_v, String'("T = "));
Std.TextIO.Write(L_v, now);
Std.TextIO.Write(L_v, String'(" S_StdV = "));
IEEE.Std_Lock_TextIO.Write(L_v, S_StdV);
Std.TextIO.WriteLine(Output, L_v);

S_StdV <= "111010101010101011100000111000111";
wait for 10 ns;

-- Method 2 -- ASSERT
assert False
report "T = " & Image(now) & " S_BitV = " & Image(S_BitV) &
HexImage(S_BitV)
severity Note;

assert False
report "S_Bit=" & Image(S_Bit) &
"; S_Int=" & Image(S_Int) &
"; S_Real=" & Image(S_Real) &
"; S_uStd=" & Image(S_uStd) &
"; S_Std =" & Image(S_Std) &
"; S_StdV=" & Image(S_StdV)
severity Note;

assert False
report "S_StdV=" & HexImage(S_StdV)
severity Note;
wait;
end process Test_Lbl;
end StringOut_a;

```

Use of the HexImage  
function for HEX display

**Figure 5.2-1a Strings to Screen, VHDL'87 and VHDL'93  
package\strgnout.vhd**

```

vsim stringout stringout_a
...
run 200
# # T = 152 ns S_Bit = 1
# # T = 152 ns S_BitV = 000011101011010
# # Thex = 152 ns S_BitV = 0F5A
# # T = 152 ns S_Int = 25
# # T = 152 ns S_Real = 3.141600e+000
# # T = 152 ns S_uStd = H
# # T = 152 ns S_Std = L
# # T = 152 ns S_StdV = 111010HLXU00001111HHHHXXXXHHHLLL1
# ** Note: T = 162 ns S_BitV = 0000111010110100F5A
# Time: 162 ns Iteration: 0 Instance:/
# ** Note: S_Bit=1 ; S_Int=25 ;S_Real=3.141600e+000
;S_uStd=H ;S_Std =L ;S_StdV=1110101010101011100000111000111
# Time: 162 ns Iteration: 0 Instance:/
# ** Note: S_StdV=3AAABC1C7
# Time: 162 ns Iteration: 0 Instance:/

```

**Figure 5.2-1b Strings to Screen, VHDL'87 and VHDL'93  
package\strgnout.txt**

```
-- Use VHDL'93
library IEEE;
use IEEE.Std_logic_1164.all;

library Work;
use Std.TextIO.all;

architecture StringOut_a of StringOut is
    ...
    -- Method 2 -- ASSERT
    assert False
        report "T = " & Time'Image(now) & " S_BitV = " &
            Bit'Image(S_Bit) & " S_Std = " &
            -- Bit_Vector'Image(S_BitV) & " S_StdV = " &
            Std_Logic'Image(S_Std)
            -- Std_Logic_Vector'Image(S_StdV)
    severity Note;
    wait;
end process Test_Lbl;
end StringOut_a;
```

**Figure 5.2-2 Use of 'Image in VHDL'93 has Limitations  
(models\strgimg.vhd)**

### 5.2.1 Writing to One file from Multiple Processes

If it is required to write onto a common file from multiple processes, then the file declaration must be defined at the top level architecture or in a package declaration. Figure 5.2.1 demonstrates the concept.

```
library STD;
use STD.TextIO.all;
entity File2 is
    generic (File_g : String := "R.out");
end File2;

architecture File2_a of File2 is
    file DataOut_f : Text is out File_g;
begin
    File1_Lbl : process
        variable MyLine_v : TextIO.Line;
    begin -- process ReadFile_Lbl
        Write(MyLine_v, now);
        Write(MyLine_v, String'("Process 1 data "));
        WriteLine(DataOut_f, MyLine_v);
        wait for 100 ns;
    end process File1_Lbl;

    File2_Lbl : process
        variable MyLine_v : TextIO.Line;
    begin -- process ReadFile_Lbl
        Write(MyLine_v, now);
        Write(MyLine_v, String'("Process 2 data "));
        WriteLine(DataOut_f, MyLine_v);
        wait for 130 ns;
    end process File2_Lbl;
end File2_a;
```

```
-- Simulation results (file "R.out")
-- 0 nsProcess 2 data
-- 0 nsProcess 1 data
-- 100 nsProcess 1 data
-- 130 nsProcess 2 data
-- 200 nsProcess 1 data
-- 260 nsProcess 2 data
-- 300 nsProcess 1 data
-- 390 nsProcess 2 data
-- 400 nsProcess 1 data
-- 500 nsProcess 1 data
-- 520 nsProcess 2 data
-- 600 nsProcess 1 data
```

**Figure 5.2.1 Writing to One file from Multiple Processes (package\file2.vhd)**

### 5.3 MULTIPLE INPUT SIGNATURE REGISTER (MISR)

**Q** | What is a MISR and how are they used?

**A** | A MISR is a Multiple Input Signature Register. The MISR is based on the same principle as *Built-In Self Test* (BIST), but it is used for simulation rather than for on-chip self test. A MISR is placed external to the a Unit Under Test (UUT) and is used in regression tests (see section 8.3). Data is strobed into the MISR when the data is stable, typically at a clock edge.

Sandi Habinc from the European Space Agency made available a MISR package in VHDL. The package declaration is shown in figure 5.3. (The MISR package is available on disk.) File *package\mistr\_tst.vhd* provides a test for the MISR package. See section 8.3 for a complete application example.

```
=====
-- Design units : MISR_Definition (Package declaration and body)
-- File name   : misr_pb.vhd
-- Purpose     : This packages defines a MISR subprogram to be used for
--                 verification of models.
-- Note        : (see further below)
-- Limitations : None known
-- Errors      : None known
-- Library     : (independent)
-- Dependencies: IEEE.Std_Loic_1164
-- Author      : Sandi Habinc, sandi@ws.estec.esa.nl
--                 ESTEC Microelectronics and Technology Section (WSM)
--                 P.O. Box 299
--                 2200 AG Noordwijk
--                 The Netherlands
-- Copyright   : European Space Agency (ESA) 1996. No part may be reproduced
--                 in any form without the prior written permission of ESA.
-- Simulator   : Synopsys v. 3.3b, on Sun SPARCstation 5, SunOS 4.1.3_U1
-----
-- Revision list
-- Version Author Date      Changes
--
-- 0.1      SH      1 July 95 New package
-- 0.2      SH      1 May  96 MISR made independent of bit ordering of inputs
-- Available at: http://www.estec.esa.nl/wsmwww/vhdl
```

```

=====
-- Multiple-Input Signature Register (MISR)10
-- This procedure implements a variable length MISR. The length is determined by
-- the length of Input, ranging from 4 to 100. The Input can be sampled on
-- either Clk edge or both, delayed by Sense, selected with the Rising and
-- Falling parameters. If neither option is selected, events on Input will
-- determine the sampling point. Events happening in the same simulation cycle,
-- differing only in delta cycles, will be sampled when the last event has
-- occurred, and the MISR will then be shifted.
--
-- The rising edge of the Reset input will reset the MISR to all-ones.
-- When sampling is made with Clk, it will re-start on the next relevant Clk
-- edge after the rising Reset edge. If Reset is detected between a Clk edge
-- and the sampling point, the MISR will be reset and the sample will be
-- ignored. If there is a relevant Clk edge in the same delta cycle as the
-- rising Reset edge, the sample will be ignored. But if the relevant Clk edge
-- is in a delta cycle after the Rising Reset edge, then the sample will be
-- made. When asynchronous sampling is used, the first event on Input after the
-- rising Reset edge (in the next or following delta cycles) will trigger the
-- first sample and shift the MISR. The MISR signal can be read at any point and
-- should be compared with a predetermined signature.
--
-- The MISR is implemented as a primitive polynomial with up to five terms. The
-- terms are taken from the text book Built-In Test for VLSI: Pseudorandom
-- Techniques, by Bardell et al. The elements in the Input vector are expanded
-- to four bits, each Std_Logic value having a unique bit pattern,
-- the intermediate vector is then divided in four and each part is shifted into
-- the MISR separately. The procedure can be used as a concurrent subprogram,
-- not needing any surrounding process or block.
--
-- The MISR is shifted from left to right, independently of falling or rising
-- bit order definition (i.e. to or downto).
--
-- Inputs: Clk, sample clock used with Rising and Falling
--          Reset, reset of MISR when an event is detected and Reset is True
--          Input, input vector to the MISR, same length and order as MISR
--          Rising/Falling:
--          False  False  sample at each Input event
--          False  True   sample Input after Sense on falling Clk edge
--          True   False  sample Input after Sense on rising Clk edge
--          True   True   sample Input after Sense on rising or falling Clk edge
--          Sense, positive time after the Clk edge when Input is sampled
--          HeaderMsg, message header
-- In/Outs: MISR, Multiple-Input Signature Register
-----

library IEEE;
use IEEE.Std_Logic_1164.all;

package MISR_Definition is
  procedure MISR(
    signal Clk:      in  Std_ULogic;           -- Sample clock
    signal Reset:    in  Boolean;                -- Reset of MISR
    signal Input:    in  Std_Logic_Vector;       -- Input vector
    signal MISR:     inout Std_Logic_Vector;     -- MISR
    constant Rising: in  Boolean := True;        -- See above
    constant Falling: in  Boolean := False;       -- See above
    constant HeaderMsg: in  String := "MISR:";   -- Message header
    constant Sense:   in  Time      := 0 ns);     -- Sense time after clock
end MISR_Definition; ===== End of package header =====

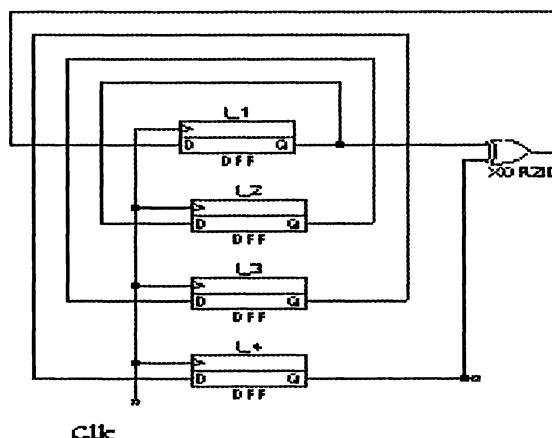
```

**Figure 5.3 MISR Package Declaration (Package body is in the file  
(package\misr\_pb.vhd))**

## 5.4 DESIGN OF A LINEAR FEEDBACK SHIFT REGISTER (LFSR)

**Q** What is a Linear Feedback Shift Register (LFSR)? Where is it used? How is it designed? Can it be synthesized into hardware?

**A** An LFSR is a very useful feedback shift register. It is used to generate pseudo-random numbers for chip Built-In-Self-Test (BIST) and for testbenches. LFSRs are particularly practical in the generation of pseudo-random numbers of type *Bit\_Vector*, *Std\_Logic\_Vector*, *Unsigned* or *Signed* types. They are preferred over the use of *Integer* type because very wide ranges can be generated (e.g., 100+ bits). Figure 5.4-1 demonstrates a hardware implementation of a 4 bit LFSR. Figure 5.4-2 shows an LFSR declaration package for the LFSR function. Figure 5.4-3 represents a portion of the package body for this package (the complete body is on the disk). Figure 5.4-4 demonstrates the design of a four-bit LFSR register. Figure 5.4-5 demonstrates the results of synthesis for this LFSR design. Figure 5.4-6 represents a testbench for the LFSR function. Chapter 8 provides a discussion of application of LFSRs used in design verification.



**Figure 5.4-1 Four Bit LFSR**

Note that other versions of the LFSR package are provided on disk to accommodate the following types:

TYPE	FILE ON DISK	COMMENTS
<code>Std.Standard.Bit_Vector</code>	<code>package\lfsrbit.vhd</code>	
<code>IEEE.Numeric_Std.Signed</code>	<code>package\lfsrnstd.bhd</code>	
<code>IEEE.Numeric_Std.Unsigned</code>	<code>package\lfsrnstd.bhd</code>	
<code>IEEE.Numeric_Bit.Signed</code>	<code>package\lfsrnbit.vhd</code>	
<code>IEEE.Numeric_Bit.Unsigned</code>	<code>package\lfsrnbit.vhd</code>	
<code>Lib.Std_Logic_Arith.Signed</code>	<code>package \lfsrarith.vhd</code>	
<code>Lib.Std_Logic_Arith.Unsigned</code>	<code>package \lfsrarith.vhd</code>	Lib is any user defined library

```

-- File name   : lfsrstd.vhd
-- Title       : Linear Feedback Shift Register Package
-- Description : for registers of lengths 2 through 64
--               : and 100, 132, 164, 200, 300 bits.
--               : Initial actual value should be a not zero vector.
--               : If is it, this function will return the value of 1.
--               : Data should NOT contain any of the following values
--               : 'U' | 'X' | 'Z' | 'W' | '-'
--               : Actual parameter can be either a variable or a
--               : signal. Size of vector can be between
--               : 2 to 64, 100, 132, 164, 200, or 300.
--               : Function detects size of actual and returns
--               : a value of the same size as
--               : the actual parameter. Example of application:
--               : signal R32_s : Std_Logic_Vector(31 downto 0) :=
--               : "0010110001010111110001010100110";
--               :
--               :      ...
--               :      R32_s <= LFSR(R32_s);
--               :
-- Source of   : "Built-In Test for VLSI: Pseudorandom Techniques",
-- equations   : Paul H. Bardell, William H. McAnney, Jacob Savir,
--               : John Wiley and Sons, 1987. [10]
--               : Equations are described in Appendix, and provide
--               : the polynomials for degrees 2 through 300.
--               :
-- Synthesis   : This code is synthesizable. Pragmas are written
--               : for Synopsys synthesizer to ignore the assert
--               : statement written for error detection
-----
library IEEE;
use IEEE.Std_Logic_1164.all;

-- pragma translate_off
Library Work;
-- for conversion of Std_Logic_Vector to a string
use Work.Image_Pkg.all;

-- pragma translate_on
package LfsrStd_Pkg is
    function LFSR(S : Std_Logic_Vector)  return Std_Logic_Vector;
    function LFSR(S : Std_uLogic_Vector) return Std_uLogic_Vector;
end LfsrStd_Pkg;

```

**Figure 5.4-2 LFSR Declaration Package (package\lfsrstd.vhd).**

```

package body LfsrStd_Pkg is
    function LFSR(S : Std_Logic_Vector) return Std_Logic_Vector is
        variable S_v   : Std_Logic_Vector(1 to S'Length);
        constant S_c   : Std_Logic_Vector(1 to S'Length)
                      := (others => '0');
    begin
        -- pragma translate_off
        S_v := To_X01(S); -- function is in Std_Logic_1164 package
        if Is_X(S_v) then
            assert False
                report "Passed parameter contains one of the following " &
                    "characters 'U' | 'X' | 'Z' | 'W' | '-' "
                severity Warning;
            assert False

```

```

report "Data passed = " &
      Work.Image_Pkg.Image(S)
severity Note;
return S;      -- Return unchanged value
end if;
-- pragma translate_on
S_v := S;
if S_v = S_c then
  return (S_c(1 to S_c'length - 1) & '1');
else
  case S'Length is
    when 2 => -- X^2 + X^1 + 1
      return (S_v(2) xor S_v(1)
              ) & S_v(1 to S'Length - 1);

    when 3 => -- X^3 + X^1 + 1
      return (S_v(3) xor S_v(1)
              ) & S_v(1 to S'Length - 1);

    when 4 => -- X^4 + X^1 + 1
      return (S_v(4) xor S_v(1)
              ) & S_v(1 to S'Length - 1);
    ...
    when 32 => -- X^32 + X^28 + X^27 + X^1 + 1
      return (S_v(32) xor S_v(28) xor
              S_v(27) xor S_v(1)
              ) & S_v(1 to S'Length - 1);

    ...
    when 64 => -- X^64 + X^4 + X^3 + X^1 + 1
      return (S_v(64) xor S_v(4) xor
              S_v(3) xor S_v(1)
              ) & S_v(1 to S'Length - 1);

    when 100 => -- X^100 + X^37 + 1
      return (S_v(100) xor S_v(37)
              ) & S_v(1 to S'Length - 1);

    when 132 => -- X^132 + X^29 + 1
      return (S_v(132) xor S_v(29)
              ) & S_v(1 to S'Length - 1);

    when 164 => -- X^164 + X^14 + X^13 + X^1 + 1
      return (S_v(164) xor S_v(14) xor
              S_v(13) xor S_v(1)
              ) & S_v(1 to S'Length - 1);
    when 200 => -- X^200 + X^163 + X^2 + X^1 + 1
      return (S_v(200) xor S_v(163) xor
              S_v(2) xor S_v(1)
              ) & S_v(1 to S'Length - 1);

    when 300 => -- X^300 + X^7 + 1
      return (S_v(300) xor S_v(7)
              ) & S_v(1 to S'Length - 1);

    when others =>
      -- pragma translate_off

```

```

        assert False
            report "Length of vector is NOT in proper range"
            severity Warning;
-- pragma translate_on
        return S_v;
    end case;
end if;
end LFSR;

function LFSR(S : Std_uLogic_Vector) return Std_uLogic_Vector is
begin
    return To_StdULogicVector(LFSR(To_StdLogicVector(S)));
end LFSR;
end LfsrStd_Pkg;

```

**Figure 5.4-3 LFSR Package Body (package\lfsrstd.vhd).**

```

library IEEE;
use IEEE.Std_Logic_1164.all;

library Work;
use Work.LfsrStd_Pkg.all;

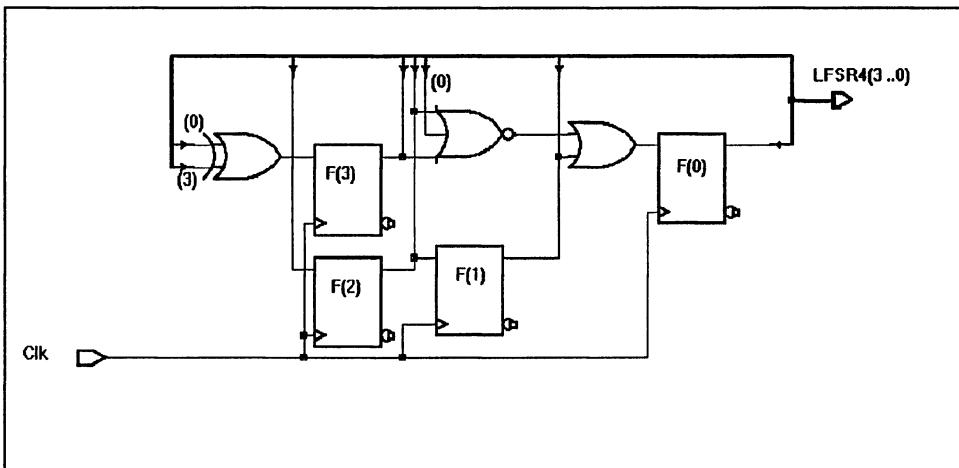
entity TestLFSR is
    port(Clk      : in  Std_Logic;
         LFSR4   : out Std_Logic_Vector(3 downto 0));
end TestLFSR;

architecture TestLFSR_a of TestLFSR is
    signal LFSR4_s : Std_Logic_Vector(3 downto 0);
begin
    LFSR_Lbl: process
    begin
        wait until Clk = '1';
        LFSR4_s <= LFSR(LFSR4_s);
    end process LFSR_Lbl;

    LFSR4 <= LFSR4_s;
end TestLFSR_a;

```

**Figure 5.4-4 Design of a Four bit LFSR Register (package\lfsr4.vhd)**



**Figure 5.4-5 Synthesized Design of a Four bit LFSR Register**

This synthesized four-bit LFSR is slightly different from the design shown in figure 5.4-1. It includes a reset to a value of ONE if the inputs are all zeros.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
library Work;
use Work.LfsrStd_Pkg.all;

entity TestLFSR is
end TestLFSR;
architecture TestLFSR_a of TestLFSR is
  signal R32_s : Std_Logic_Vector(31 downto 0) := 
    "0010110001010111110001010100110";
  signal R16_s : Std_Logic_Vector(0 to 15) := "0100110U111Z0100";
  signal R6_s : Std_Logic_Vector(0 to 5) := (Others => '0');
  signal R8_s : Std_Logic_Vector(7 downto 0);
  signal R4_s : Std_Logic_Vector(3 downto 0) := "HL10";
begin
  R32_s <= LFSR(R32_s) after 10 ns;
  R16_s <= LFSR(R16_s) after 10 ns;
  R6_s <= LFSR(R6_s) after 10 ns;
  R4_s <= LFSR(R4_s) after 10 ns;
  LFSR_Lbl: process
    variable R8_v : Std_Logic_Vector(7 downto 0) := "10101110";
  begin
    R8_v := LFSR(R8_v);
    R8_s <= R8_v;
    wait for 10 ns;
  end process LFSR_Lbl;
end TestLFSR_a;

```

**Figure 5.4-6 Testbench for the LFSR function (package\lfsr\_tb.vhd)**

## 5.5 RANDOM NUMBER GENERATION

**Q** | What are the methods to generate random numbers of type *Integer*, *Real*, *Bit\_Vector* and *Std\_Logic\_Vector*?

**A** | The package *RandomInt\_Pkg* shown in figure 5.5-1 provides a simple means to generate integer random numbers. Figure 5.5-2 represents an application model of this package. Note that the random numbers generated by the procedure *Random* is very sensitive to the parameters passed to the procedure. The user is advised to experiment with those parameters using either simulation or spreadsheet analysis.

```
package RandomInt_Pkg is
    procedure Random(variable Seed_v      : inout Integer;
                      constant Modulus_c : in     Integer);
end RandomInt_Pkg;

package body RandomInt_Pkg is
    procedure Random(variable Seed_v      : inout Integer;
                      constant Modulus_c : in     Integer) is
        constant Multiplier_c : Integer := 25173;
        constant Increment_c  : Integer := 13849;
    begin
        Seed_v := (Multiplier_c * Seed_v + Increment_c) mod Modulus_c;
    end Random;
end RandomInt_Pkg;
```

Figure 5.5-1 Integer Random Package (package\rndm\_int.vhd)

```
architecture Random_a of Random is -- empty entity
begin
    signal Random_s : Integer;
    PseudoRandomLbl: process
        variable Seed_v      : Integer := 17654;
        constant Modulus_c   : Integer := 65536; -- rollover value
    begin
        Work.RandomInt_Pkg.Random(Seed_v, Modulus_c); <- Providing the full path for
        Random_s <= Seed_v;                                the procedure enhances
        wait for 10 ns; --repeat process every 10 ns      maintainability and
        end process PseudoRandomLbl;                         readability
    end Random_a;
```

Figure 5.5-2 Application of Integer Random Package (package\rand\_ea.vhd)

The package *RND* in file *package\rndm\_pb.vhd* provides a broad range of random functions. The package is available on disk, and the reader is encouraged to view this file for more details. For pseudo-random numbers in *Bit\_Vector* and *Std\_Logic\_Vector*, the reader is encouraged to use the LFSR package described in section 5.4. The bit vector length determines the range of the pseudo-random number. A conversion function is used to convert the bit vector to an integer. Section 1.10 provides an application for the use of the LFSR package for random number generation.

## 5.6 DEFERRED CONSTANT IN PACKAGE DECLARATION

**Q** | How come a deferred constant cannot be used to define an expression or a subtype in the same package declaration where the constant is declared?. For example, the code shown in figure 5.6 the compiler yields an error on line 4 for package *T\_Pkg*, but yet, no error occurs for package *T2\_Pkg* and package *T3\_Pkg*.

```

package T_Pkg is
    constant X_c : Natural;

    subtype X_Typ is Integer range 0 to X_c; -- Error ??
    -- vcom H:/BEN/TEST2.VHD
    -- # -- Loading package standard
    -- # -- Compiling package t_pkg
    -- # ERROR: H:/BEN/TEST2.VHD(4):
    --       Illegal use of deferred constant x_c.

end T_Pkg;

package body T_Pkg is
    constant X_c : Natural := 4;
end T_Pkg;

package T2_Pkg is
    constant X_c : Natural;
end T2_Pkg;

library Work;
use Work.T2_Pkg.all;
package T3_Pkg is
    subtype X_Typ is Integer range 0 to X_c;
end T3_Pkg;
```



**Figure 5.6 Referencing Deferred Constant deferrpb.vhd**

**A** | The IEEE-Std-1076-1993 LRM, Section 2.6, page 30, line 453-457 states that [1] *the result of evaluating an expression that references a deferred constant before the elaboration of the corresponding full declaration is not defined in the language.* The reason this code does not work in *T\_Pkg* is that the constant can never be defined at the time the subtype is elaborated. Specifically, the value for *X\_c* is not known until the package body is elaborated later. However, in the other case, when *T3\_pkg* is elaborated, the value for *X\_c* is known, because the *T2\_pkg* has already been elaborated and the constant has been defined.

## 5.7 COMPLEX NUMBERS AND OVERLOADED OPERATORS

**Q** | How can complex numbers and overloaded operators for those numbers be defined? How can multiplication of two two-by-two complex matrices be modeled?

**A** | A complex number consists of a *real* field and an *imaginary* field. A record can be used to define the a complex type as shown in figure 5.7-1. The addition operators operate on the corresponding fields of the complex numbers. For example, the addition of two complex numbers yields the addition of the *real* fields for the *real* result, and the addition of the *imaginary* fields for the *imaginary* result. The multiplication of two two-by-two matrices is defined in table 5.7. The implementation is defined in the package

*Complex\_Pkg* where each matrix is stored as an array of the matrix record. Figure 5.7-2 demonstrates an application example of this package, while figure 5.7-3 provides the resulting file provided by simulation.

**Table 5.7 Complex Matrix Multiplication**

Matrix A	Matrix B	$C = A * B$	
a1      a2	b1      b2	(a1*b1 + a2*b3)	(a1*b2 + a2*b4)
a3      a4	b3      b4	(a3*b1 + a4*b3)	(a3*b2 + a4*b4)

```

library IEEE;
use IEEE.Std_Loic_1164.all;
use IEEE.Std_Loic_Arith.all;

package Complex_Pkg is
    constant Length_c : Integer := 32;
    type Complex32_Typ is record
        R : Signed(Length_c - 1 downto 0); ← Complex type definition
        I : Signed(Length_c - 1 downto 0);
    end record;

    type Complex64_Typ is record
        R : Signed(2 * Length_c - 1 downto 0);
        I : Signed(2 * Length_c - 1 downto 0); ← Complex 2x2 array type definition
    end record;

    type A2x2C32_Typ is array(1 to 4) of Complex32_Typ;

    type A2x2C64_Typ is array(1 to 4) of Complex64_Typ;

    function "+" (A : Complex32_Typ;
                  B : Complex32_Typ) return Complex32_Typ;

    function "-" (A : Complex32_Typ;
                  B : Complex32_Typ) return Complex32_Typ;

    function "+" (A : Complex64_Typ;
                  B : Complex64_Typ) return Complex64_Typ;

    function "-" (A : Complex64_Typ;
                  B : Complex64_Typ) return Complex64_Typ;

    function "*" (A : Complex32_Typ;
                  B : Complex32_Typ) return Complex64_Typ; ← Operator for complex
                                                multiplication operator.

    function "*" (A : A2x2C32_Typ;
                  B : A2x2C32_Typ) return A2x2C64_Typ;

    function "+" (A : A2x2C32_Typ;
                  B : A2x2C32_Typ) return A2x2C32_Typ;

    function "-" (A : A2x2C32_Typ;
                  B : A2x2C32_Typ) return A2x2C32_Typ;
end Complex_Pkg;
=====
```

```

package body Complex_Pkg is
    function "+" (A : Complex32_Typ;
                  B : Complex32_Typ) return Complex32_Typ is
        variable V : Complex32_Typ;
    begin
        V.R := A.R + B.R;
        V.I := A.I + B.I;
        return V;
    end "+";

    function "-" (A : Complex32_Typ;
                  B : Complex32_Typ) return Complex32_Typ is
        variable V : Complex32_Typ;
    begin
        V.R := A.R - B.R;
        V.I := A.I - B.I;
        return V;
    end "-";

    function "+" (A : Complex64_Typ;
                  B : Complex64_Typ) return Complex64_Typ is
        variable V : Complex64_Typ;
    begin
        V.R := A.R + B.R;
        V.I := A.I + B.I;
        return V;
    end "+";

    function "-" (A : Complex64_Typ;
                  B : Complex64_Typ) return Complex64_Typ is
        variable V : Complex64_Typ;
    begin
        V.R := A.R - B.R;
        V.I := A.I - B.I;
        return V;
    end "-";

    function "*" (A : Complex32_Typ;
                  B : Complex32_Typ) return Complex64_Typ is
        variable V : Complex64_Typ;
    begin
        V.R := (A.R * B.R) - (A.I * B.I);
        V.I := (A.I * B.R) + (A.R * B.I);
        return V;
    end "*";

    function "*" (A : A2x2C32_Typ;
                  B : A2x2C32_Typ) return A2x2C64_Typ is
        variable V : A2x2C64_Typ;
    begin
        V(1) := (A(1) * B(1)) + (A(2) * B(3));
        V(2) := (A(1) * B(2)) + (A(2) * B(4));
        V(3) := (A(3) * B(1)) + (A(4) * B(3));
        V(4) := (A(3) * B(2)) + (A(4) * B(4));
        return V;
    end "*";

    function "+" (A : A2x2C32_Typ;
                  B : A2x2C32_Typ) return A2x2C32_Typ is
        variable V : A2x2C32_Typ;
    begin
        for I in A'range loop
            V(I) := A(I) + B(I);
        end loop;

        return V;
    end "+";

```

← Multiplication of a 2x2 complex matrices

```

function "-" (A : A2x2C32_Typ;
              B : A2x2C32_Typ) return A2x2C32_Typ is
    variable V : A2x2C32_Typ;
begin
    for I in A'range loop
        V(I) := A(I) - B(I);
    end loop;

    return V;
end "-";
end Complex_Pkg;

```

Figure 5.7-1 Complex Package (package\complexp.vhd)

```

library Std;
use Std.TextIO.all;

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_Arith.all;
use IEEE.Std_Logic_TextIO.all;

library Work;
use Work.Complex_Pkg.all;
use Work.Image_Pkg.all;

entity Complex is
end Complex;

architecture Complex_a of Complex is
    signal A_s : A2x2C32_Typ; -- 2x2 complex matrix
    signal B_s : A2x2C32_Typ;
    signal C_s : A2x2C64_Typ;

begin -- Complex_a
    -- Process: Complex_Lbl
    -- Purpose: Test the 2x2 complex matrix multiplication
    Complex_Lbl : process
        variable A : A2x2C32_Typ :=
            (1 => (R => Conv_Signed(1, 32), -- to 32 bits
                    I => Conv_Signed(2, 32)),
             2 => (R => Conv_Signed(3, 32),
                    I => Conv_Signed(4, 32)),
             3 => (R => Conv_Signed(5, 32),
                    I => Conv_Signed(6, 32)),
             4 => (R => Conv_Signed(7, 32),
                    I => Conv_Signed(8, 32)));
        variable B : A2x2C32_Typ :=
            (1 => (R => Conv_Signed(10, 32), -- to 32 bits
                    I => Conv_Signed(11, 32)),
             2 => (R => Conv_Signed(12, 32),
                    I => Conv_Signed(13, 32)),
             3 => (R => Conv_Signed(14, 32),
                    I => Conv_Signed(15, 32)),
             4 => (R => Conv_Signed(16, 32),
                    I => Conv_Signed(17, 32)));
        variable C : A2x2C64_Typ;
        variable L : Line;
        file Result_f : Text is out "result.txt"; -- VHDL'87
    begin
        for I in A'range loop
            for J in A'range loop
                A(I, J) := (R => Conv_Signed(1, 32),
                            I => Conv_Signed(2, 32));
                B(I, J) := (R => Conv_Signed(10, 32),
                            I => Conv_Signed(11, 32));
            end loop;
        end loop;
        for I in A'range loop
            for J in A'range loop
                C(I, J) := A(I, J) * B(I, J);
            end loop;
        end loop;
        for I in A'range loop
            for J in A'range loop
                L := Line'Value(C(I, J));
                TextIO.Put(Result_f, L);
            end loop;
        end loop;
        TextIO.Close(Result_f);
    end;
end process;

```

Initialize matrix with 32 bit numbers

```

begin -- process Complex_Lbl
  C := A * B;                                ← Use of overloaded multiplication operator
  Write(L, String'("Multiplication"));
  WriteLine(Result_f, L);

  WriteA_Lbl : for I in 1 to 4 loop
    Write(L, I);
    Write(L, " A.r = " & HexImage(A(I).R) & " A.I = " & HexImage(A(I).I));
    WriteLine(Result_f, L);
    Write(L, " A.r = " & DeclImage(A(I).R) & " A.I = " & DeclImage(A(I).I));
    WriteLine(Result_f, L);
  end loop WriteA_Lbl;                         ← Writing operators to file. Use of HexImage
                                                and DeclImage functions for conversion of
                                                signed number to hex and decimal characters

  WriteLine(Result_f, L);

  WriteB_Lbl : for I in 1 to 4 loop
    Write(L, I);
    Write(L, " B.r = " & HexImage(B(I).R) & " B.I = " & HexImage(B(I).I));
    WriteLine(Result_f, L);
    Write(L, " B.r = " & DeclImage(B(I).R) & " B.I = " & DeclImage(B(I).I));
    WriteLine(Result_f, L);
  end loop WriteB_Lbl;

  WriteLine(Result_f, L);

  WriteC_Lbl : for I in 1 to 4 loop
    Write(L, I);
    Write(L, " C.r = " & HexImage(C(I).R) & " C.I = " & HexImage(C(I).I));
    WriteLine(Result_f, L);
    Write(L, " C.r = " & DeclImage(C(I).R) & " C.I = " & DeclImage(C(I).I));
    WriteLine(Result_f, L);
  end loop WriteC_Lbl;

  WriteLine(Result_f, L);
  A_s <= A;
  B_s <= B;
  C_s <= C;
  wait for 100 ns;

  C32 := A + B;
  Write(L, String'("Addition"));
  WriteLine(Result_f, L);

  WriteC32_Lbl : for I in 1 to 4 loop
    Write(L, I);
    Write(L, " C32.r = " & HexImage(C32(I).R) & " C32.I = " & HexImage(C32(I).I));
    WriteLine(Result_f, L);
    Write(L, " C32.r = " & DeclImage(C32(I).R) & " C32.I = " & DeclImage(C32(I).I));
    WriteLine(Result_f, L);
  end loop WriteC32_Lbl;

  WriteLine(Result_f, L);
  wait;
end process Complex_Lbl;
end Complex_a;

```

**Figure 5.7-2 Application Example of the Complex Package  
(\package\complex.vhd)**

```

Multiplication
1 A.r = 00000001 A.I = 00000002
  A.r = 1                               A.I = 2
2 A.r = 00000003 A.I = 00000004
  A.r = 3                               A.I = 4
3 A.r = 00000005 A.I = 00000006
  A.r = 5                               A.I = 6
4 A.r = 00000007 A.I = 00000008
  A.r = 7                               A.I = 8

1 B.r = 0000000A B.I = 0000000B
  B.r = 10                             B.I = 11
2 B.r = 0000000C B.I = 0000000D
  B.r = 12                             B.I = 13
3 B.r = 0000000E B.I = 0000000F
  B.r = 14                             B.I = 15
4 B.r = 00000010 B.I = 00000011
  B.r = 16                             B.I = 17

1 C.r = FFFFFFFFFFFFFFE2 C.I = 0000000000000084
  C.r = -30                           C.I = 132
2 C.r = FFFFFFFFFFFFFFDE C.I = 0000000000000098
  C.r = -34                           C.I = 152
3 C.r = FFFFFFFFFFFFFFDA C.I = 000000000000014C
  C.r = -38                           C.I = 332
4 C.r = FFFFFFFFFFFFFFD6 C.I = 0000000000000180
  C.r = -42                           C.I = 384

Addition
1 C.r = 0000000B C.I = 0000000D
  C.r = 11                           C.I = 13
2 C.r = 0000000F C.I = 00000011
  C.r = 15                           C.I = 17
3 C.r = 00000013 C.I = 00000015
  C.r = 19                           C.I = 21
4 C.r = 00000017 C.I = 00000019
  C.r = 23                           C.I = 25

```

**Figure 5.7-2 Simulation Results**

## 5.8 IEEE STANDARDS

**Q**

What are the current *IEEE VHDL* standards and statuses of these standards?

**A**

The *IEEE VHDL* packages are generally compiled in library *IEEE*. Package *Std\_Logic\_1164* is already a standard (copy is available on disk). To support synthesis, two packages (one for type *Bit* and one for type *Std\_Logic*) have passed the balloting phase. These packages include *Numeric\_Bit* and *Numeric\_Std*. It is expected that these packages will become a standard in 1997. These packages may be downloaded via *ftp* from *vhdl.org* (anonymous *login*) in subdirectory */vi/synth*. When these standards are formally published, they can be ordered through the IEEE Customer Service Center at (800) 678-IEEE. In addition, the IEEE Web site (<http://stdsbbs.ieee.org/>) provides another mean to obtain information about standards that can be ordered through the IEEE.

The published version of these packages will also include a diskette with the packages in electronic form.

These packages contain among other things:

- overloaded `+`, `-`, `*` / `rem` `mod` `>` `<` `=` `<=` `/=` `not` `and` `or` `nand` `nor` `xor` `xnor`
- `shift_left`, `shift_right`, `rotate_left`, `rotate_right`, `resize` functions
- `to_integer`, `to_unsigned`, `to_signed` functions
- `Match` function

Another standard of interest is the *IEEE Vital Standard ASIC Modeling Specification P1076.4* available through the IEEE (see phone number above). The VITAL specification and packages may also be available via *ftp* from *vhdl.org* (anonymous *login*) in subdirectory */vi/vital*.

Yet, another upcoming standard is the *IEEE Standard for VHDL Register Transfer Level Synthesis* developed by VHDL Synthesis Interoperability Working Group. The goals of this standard are to "define a means of writing VHDL that guarantees the interoperability of VHDL descriptions between register transfer level synthesis tools that comply to the standard." It is expected that this standard will be officially available from the IEEE in 1998.

## **6. MODELS**

---

---

This section defines the modeling of two separate classes of models: memory and switch. These models include:

1. **Memory Model.** The traditional memory model, where all the information is stored in a variable, is demonstrated. In addition, the modeling of very large memories (GBytes) is illustrated using a cache like approach and a dynamic or paging method.
2. **Wire Model.** This is a model of a zero ohm resistor (bridge or pass-through component) that exemplifies the concept of a break before make. It is a good learning example because it demonstrates the concepts of drivers, delta times, and the *generate* statement. Many users have requested this model because it reflects real hardware interconnections.
3. **Error Injector Model.** The wire model was modified to create an error injector, where a third port is used to identify the type of stuck-at (or none) fault to apply at the pass through switch.
4. **Transfer Gate Model.** This model represents a transistor switch. It represents a modification of the error injector model.

These models demonstrate the following VHDL concepts:

- Drivers
- records
- linked lists
- Delta time
- Generate statement

## 6.1 LARGE RAM MODEL FOR SIMULATION.

**Q**

Large memory models (e.g., with a thirty-two or sixty-four bits addressing range) require a large amount of host memory, and are slow when the simulator performs disk swapping. How can large memories be modeled efficiently?

**A**

Creating a very large non-synthesizable memory bus functional model (BFM) causes a problem when the memory is declared in the traditional method as shown below (see section 9.1.7 for a further discussion of this issue):

```
Memory_Typ is array(Natural'range) of Std_Logic_Vector(31 downto 0);
```

Generally, in testbench applications, not ALL the memory locations are used during a simulation. The purpose of the memory model is to verify that the unit under test (UUT) interfaces properly with the memory. Typically, various pages or regions of the memory are addressed. The following methods can be used to model a memory.

1. Traditional memory modeling
2. Efficient memory modeling
3. Fixed array caching
4. Fixed page caching
5. Dynamic page caching
6. Disk paging with swap files
7. C Language Interface (if the compiler provides that feature)

Each of these methods is discussed in the following subsections. A simple testbench and configuration declarations are provided to demonstrate the instantiation and utilization of the memory component.

### 6.1.1 Traditional Memory Modeling

Figure 6.1.1 demonstrates the modeling of a memory that is initialized, at startup, by a file defined in a generic (and initialized to "memdata.txt"). The storage element is a variable typed as an array of a constrained *Std\_Logic\_Vector*. The size of the constraint is extracted from generics. The memory allocation (in bytes) required by the simulator is typically represented by this equation:

$$\begin{aligned} \text{Simulation memory size} = & \text{model memory depth *} \\ & \text{model memory width *} \\ & \text{simulation memory/object storage} \end{aligned}$$

A typical object storage for a one-bit of enumerated type (e.g., *Std\_Logic*) is one byte. Therefore, a 640K x 32 bit memory model would require twenty megabytes of simulation memory. On a personal computer, this is significant because of the limited resources. See section 9.1.7 for a discussion of the impact of the size of modeling memory on simulation speed.

File *models\memrytb.vhd* (available on disk) represents a testbench for the traditional memory model.

```

--          READ      <- write at this edge
--      RdWrF   -----+ +-----+
--                  +_____
--          CeF    ---+ +---+ +-----+
--                  +---+ +-----+
library IEEE;
use IEEE.Std_Logic_1164.all;

package Mem_Pkg is
    constant DataWidth_c : Natural; -- Data width
    constant AddrWidth_c : Natural; -- Address width
    constant MaxDepth_c : Natural;

    function CharToStd(Char_v : character) return Std_Logic;
end Mem_Pkg;

package body Mem_Pkg is
    constant DataWidth_c : Natural := 8; -- deferred constant
    constant AddrWidth_c : Natural := 16; -- Address width
    constant MaxDepth_c : Natural := 2 ** AddrWidth_c; -- 64K
    function CharToStd(Char_v : character) return Std_Logic is
        begin
            case Char_v is
                when '1'      => return '1';
                when '0'      => return '0';
                when others  =>
                    return '0';
                    assert False
                    report "Input Character is NOT a 1 or 0"
                    severity Warning;
            end case;
        end CharToStd;
    end Mem_Pkg;

library IEEE;                                     -- used because Std_Logic
use IEEE.Std_Logic_1164.all;                      -- signals are resolved

library Work;
use Work.Mem_Pkg.all;
use Work.Mem_Pkg;                                ← This declaration enables the shortening of the path name by
                                                providing the package name without the library name.
                                                Providing the package name enhances readability
entity Memory_Nty is
    generic (FileName_g : String(1 to 12):= "memdata_.txt");
    port(
        MemAddr      : in  Natural;
        MemData      : inout Std_Logic_Vector
                        (Mem_Pkg.DataWidth_c - 1 downto 0);
        RdWrF       : in  Std_Logic;
        CeF         : in  Std_Logic); -- Chip Select
end Memory_Nty;

```

```
architecture Memory_a of Memory_Nty is
begin -- Memory_a
    Memory_Lbl : process(MemAddr, MemData, RdWrF, CeF)
        use Std.TextIO.all; -- localize TextIO to this process
        use Std.TextIO;

        subtype Data_Typ is
            Std_Logic_Vector(Mem_Pkg.DataWidth_c - 1 downto 0);
        subtype MemSize_Typ is Integer range 0 to Mem_Pkg.MaxDepth_c - 1;
        type Mem_Typ is array(MemSize_Typ) of Data_Typ;

        variable Memory_v : Mem_Typ;
        variable Initialization_v : Boolean := True;
        file     MemData_f : TextIO.text is in FileName_g;

        procedure MemoryData -- Preload memory from a file, VHDL'87
            (variable FileName_v : in TextIO.Text;
             variable Memory_v : inout Mem_Typ) is
            variable InLine_v : TextIO.Line;
            variable MemAddress_v : MemSize_Typ := 0; -- 0 to 64k
            variable Word_v : Data_Typ;
            variable WordIndex_v : Natural;
        begin
            File_Lbl : while not TextIO.Endfile(FileName_v) loop
                -- Read 1 line from the input file
                TextIO.ReadLine(FileName_v, InLine_v);
                -- Test if InLine_v is an empty line,
                next File_Lbl when InLine_v'length = 0; -- null line
                if InLine_v'length < DataWidth_c then -- error
                    assert False
                        report "Data width in file is less than defined width"
                        severity Warning;
                    next File_Lbl;
                -- detect lines of 80 characters with spaces
                elsif InLine_v(DataWidth_c) = ' ' then -- error
                    assert False
                        report "Data in file has a space, width too small"
                        severity Warning;
                    next File_Lbl;
                end if;
                -- Read a data word
                WordIndex_v := InLine_v'left;
                -- Convert characters to Std_Logic and fill a data word
                LoadWord_Lbl : for W_i in Mem_Pkg.DataWidth_c - 1 downto 0 loop
                    Word_v(W_i) := Mem_Pkg.CharToStd(InLine_v(WordIndex_v));
                    WordIndex_v := WordIndex_v + 1;
                end loop LoadWord_Lbl;
                -- Store word in memory and increment address index
                Memory_v(MemAddress_v) := Word_v;
                MemAddress_v := MemAddress_v + 1;
            end loop File_Lbl;
        end MemoryData;
    end Memory_a;
```

```

begin -- process Memory_Lbl
  -- Load memory if in initialization
  if Initialization_v then
    MemoryData(FileName_v  => MemData_f,
                Memory_v   => Memory_v);
    Initialization_v := False;
  end if;

  -- Normal operation
  if CeF = '0' then -- memory transaction
    if RdWrF'event and
      RdWrF'last_value = '0' and -- positive transition
      RdWrF = '1' then        -- of RdWrF
      Memory_v(MemAddr) := MemData; -- WRITE OPERATION
    elsif RdWrF = '1' then    -- memory read
      MemData <= Memory_v(MemAddr);
    end if;
  else
    MemData <= (others => 'Z');
  end if;
end process Memory_Lbl;
end Memory_a;

```

**Figure 6.1.1 Memory Initialized at Startup by a File (models\memory.vhd)**

### 6.1.2 Efficiency Memory Modeling

Since the RAM is only needed for simulation, it is only necessary that the I/O be of type *Std\_Logic\_Vector*. Internally the model could use an *array of Integers*. This will reduce the simulation memory requirements in the array by a significant factor. For example, an enumerated data type (such as *Std\_Logic*) typically requires one byte per element (or 32 bytes for a 32 bit vector), whereas an Integer requires four bytes. The range of *Integer* values is limited to  $-(2^{31} - 1)$  to  $(2^{31} - 1)$  (see section 1.5.1), thus limiting the memory width to 31 bits. The conversion functions from and to *Integers* are needed to convert to the proper data types. In addition, all the metalogical values (i.e., ‘U’, ‘X’, ‘W’, ‘-’) and the weak values are lost in the conversions; therefore, this approach has some limitations. Metalogical values define the behavior of the model itself rather than the behavior of the hardware being synthesized.

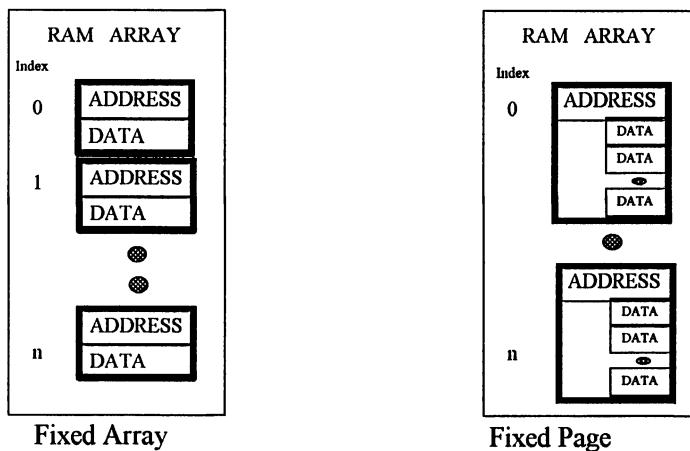
An optional approach to using the *Integer* type is the *Bit\_Vector* type. This type is sometimes faster, since some simulators pack bit vectors into integers. However, this approach is simulator dependent.

Memory elements should be declared as a variable object in a process, rather than a signal of an architecture (see section 9.1 on methods to enhance simulation speed). Only the interfaces to the memory should be declared as signals.

### 6.1.3 Fixed Array Caching

This method presupposes that a limited number of addresses, spread across a wide addressing space, will be used during simulation. This approach uses a fixed array of records, where each record stores in its fields a cached address and data line. This array is only as big as the number of addresses the user is expected to access. A variable is used

to identify the highest index pointing to the data stored in this cache-like RAM. Every transaction (READ or WRITE) must search through the range of written locations to find a match, in the address field, between the actual address and the stored address. If the transaction is a WRITE and there is no match in the address, then the transaction (address and data) is written into the array at the highest indexed address. Otherwise (there is a match), the transaction is overwritten at the address indexed by the position where previous data existed. The index is incremented if it does not exceed the limit. A READ of a non-stored memory location yields 'X's. Figure 6.1.3-1 is a pictorial view of the memory data structure for the fixed array and fixed page models. Figure 6.1.3-2 represents the code for the fixed array model.



**Figure 6.1.3-1 Fixed Array/Page Caching Models**

```

library IEEE;
use IEEE.Std_Lock_1164.all;
entity RAM is
generic(DataWidth_g : Positive := 16;
         AddrWidth_g : Positive := 32);
port(Address      : in  Std_Lock_Vector(AddrWidth_g - 1 downto 0);
      Data        : inout Std_Lock_Vector(DataWidth_g - 1 downto 0);
      CeF         : in   Std_Lock;
      RdWrF       : in   Std_Lock);
end RAM;

architecture RamFix_a of RAM is
constant CacheSize_c : Positive := 255;
subtype CacheRange_Typ is Integer range 0 to CacheSize_c;
type CacheBlock_Typ is record
  Address: Std_Lock_Vector (AddrWidth_g - 1 downto 0);
  Data   : Std_Lock_Vector (DataWidth_g - 1 downto 0);
end record;

```

```

begin
  Mem_Lbl: process(Address, Data, RdWrF, CeF)
    type RAM_Typ is array (CacheRange_Typ) of CacheBlock_Typ;
    variable IndexMax_v : CacheRange_Typ := 0;
    variable RAM_v : RAM_Typ :=
      (others => (Address => (others => 'X'),
                    Data     => (others => 'X')));
    variable FoundAddr_v : Boolean;
begin
  if (CeF = '0') or (CeF = 'L') then
    FoundAddr_v := False;
  Search_Lbl: for Cache_i in 0 to IndexMax_v loop
    if RAM_v(Cache_i).Address = Address then -- Already accessed
      FoundAddr_v := True;
    if RdWrF'event and      -- check for rising edge for a write
      ((RdWrF = '1') or (RdWrF = 'H')) and
      ((RdWrF'last_value = '0') or
       (RdWrF'last_value = 'L')) then
      RAM_v(Cache_i).Address := Address;
      RAM_v(Cache_i).Data   := Data;

    elsif ((RdWrF = '1') or (RdWrF = 'H')) then
      Data <= RAM_v(Cache_i).Data;

      -- check for falling edge for a write
    elsif RdWrF'event and
      ((RdWrF = '0') or (RdWrF = 'L')) and
      ((RdWrF'last_value = '1') or (RdWrF'last_value = 'H')) then
      null;

    else
      assert False
        report "Error in RdWrF signal"
          severity Warning;
    end if;
    exit Search_Lbl;
  end if;
end loop;

if not FoundAddr_v then
  if RdWrF'event and      -- check for rising edge for a write
    ((RdWrF = '1') or (RdWrF = 'H')) and
    ((RdWrF'last_value = '0') or (RdWrF'last_value = 'L')) then
    RAM_v(IndexMax_v).Address := Address;
    RAM_v(IndexMax_v).Data   := Data;
    if IndexMax_v /= CacheSize_c then
      IndexMax_v := IndexMax_v + 1;

    else
      assert False
        report "Maximum limit for cache entry is reached"
          severity Note;
    end if;

  elsif ((RdWrF = '1') or (RdWrF = 'H')) then
    Data <= (others => 'X');

```

```

-- check for falling edge for a write
elsif RdWrF'event and
  ((RdWrF = '0') or (RdWrF = 'L')) and
  ((RdWrF'last_value = '1') or (RdWrF'last_value = 'H')) then
  null;

else
  assert False
    report "Error in RdWrF signal"
    severity Warning;
  end if;
end if;
else -- Ce_F /= '0' | 'L'
  data <= (others => 'Z');
end if;
end process Mem_Lbl;
end RamFix_a;

```

**Figure 6.1.3 Fixed Array Caching RAM (models\ramfix.vhd)**

#### 6.1.4 Fixed Page Caching

This model is similar to the fixed array model with the exception that, for each page address, a page (instead of a word) of RAM data is made accessible. The model consists of a fixed array of RAM pages as shown in figure 6.1.4.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity RAM is
  generic(DataWidth_g : Positive := 16;
           AddrWidth_g : Positive := 32);
  port(Address      : in  Std_Logic_Vector(AddrWidth_g - 1 downto 0);
        Data         : inout Std_Logic_Vector(DataWidth_g - 1 downto 0);
        CeF          : in   Std_Logic;
        RdWrF        : in   Std_Logic);
end RAM;
architecture RamPage_a of RAM is
  constant CacheSize_c : Positive := 255;
  constant PageSize_c : Positive := 8; -- 2**8, 256 words/page
  subtype CacheRange_Typ is Integer range 0 to CacheSize_c;
  type Page_Typ is array(0 to 2**PageSize_c - 1) of
                Std_Logic_Vector(DataWidth_g - 1 downto 0);
  type CacheBlock_Typ is record
    Address: Std_Logic_Vector (AddrWidth_g - 1 downto 0);
    Page   : Page_Typ;
  end record;
  -----
  -- Function to convert an vector and return it's integer equivalent *
  -----
  function Vect2Int (Arg_c : in Std_Logic_Vector) return Integer is
    use IEEE.Numeric_Std.all;
  begin
    return IEEE.Numeric_Std.TO_INTEGER(UNSIGNED(Arg_c));
  end;

```

```

begin
  Mem_Lbl: process(Address, Data, RdWrF, CeF)
    type RAM_Typ is array (CacheRange_Typ) of CacheBlock_Typ;
    variable IndexMax_v : CacheRange_Typ := 0;
    variable RAM_v : RAM_Typ :=
      (others => (Address => (others => 'X'),
                    Page     => (others =>
                      (others =>'X'))));
    variable FoundAddr_v : Boolean;
    variable Page_v       : Natural;
begin
  Page_v := Vect2Int(Address(PageSize_c - 1 downto 0));
  if (CeF = '0') or (CeF = 'L') then
    FoundAddr_v := False;
  Search_Lbl: for Cache_i in 0 to IndexMax_v loop
    if RAM_v(Cache_i).Address(AddrWidth_g -1 downto PageSize_c) =
       Address(AddrWidth_g -1 downto PageSize_c) then
      -- Already accessed
      FoundAddr_v := True;
    if RdWrF'event and    -- check for rising edge for a write
       ((RdWrF = '1') or (RdWrF = 'H')) and
       ((RdWrF'last_value = '0') or (RdWrF'last_value = 'L')) then
      RAM_v(Cache_i).Address := Address;
      RAM_v(Cache_i).Page(Page_v) := Data;

    elsif ((RdWrF = '1') or (RdWrF = 'H')) then
      Data <= RAM_v(Cache_i).Page(Page_v);

    -- check for falling edge for a write
    elsif RdWrF'event and
       ((RdWrF = '0') or (RdWrF = 'L')) and
       ((RdWrF'last_value = '1') or (RdWrF'last_value = 'H')) then
      null;

    else
      assert False
        report "Error in RdWrF signal"
        severity Warning;
    end if;
    exit Search_Lbl;
  end if;
  end loop;
  if not FoundAddr_v then
    if RdWrF'event and    -- check for rising edge for a write
       ((RdWrF = '1') or (RdWrF = 'H')) and
       ((RdWrF'last_value = '0') or (RdWrF'last_value = 'L')) then
      RAM_v(IndexMax_v).Address := Address;
      RAM_v(IndexMax_v).Page(Page_v) := Data;
    if IndexMax_v /= CacheSize_c then
      IndexMax_v := IndexMax_v + 1;
    else
      assert False
        report "Maximum limit for cache entry is reached"
        severity Note;
    end if;
  elsif ((RdWrF = '1') or (RdWrF = 'H')) then
    Data <= (others => 'X');
  end if;
end;

```

```

-- check for falling edge for a write
elsif RdWrF'event and
  ((RdWrF = '0') or (RdWrF = 'L')) and
  ((RdWrF'last_value = '1') or (RdWrF'last_value = 'H')) then
  null;

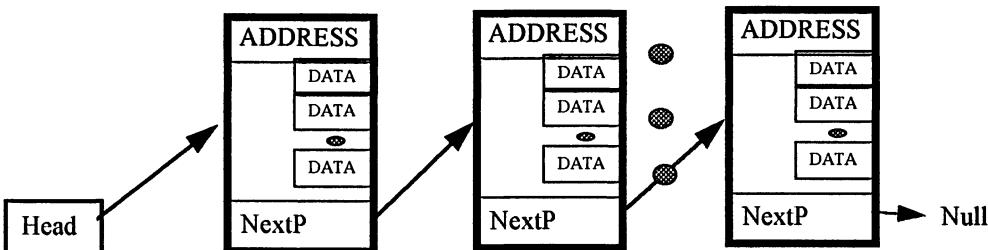
else
  assert False
    report "Error in RdWrF signal"
    severity Warning;
  end if;
end if;
else -- Ce_F /= '0' | 'L'
  data <= (others => 'Z');
end if;
end process Mem_Lbl;
end RamPage_a;

```

**Figure 6.1.4 Fixed Page Caching (models\rampage.vhd)**

### 6.1.5 Dynamic Page Caching

This solution is similar to the fixed array caching, but a linked list is used instead of an array. An object of *access* type is used to point to a record that includes: the page address of the current memory page, RAM data block for that page, and a pointer to the next element. Figure 6.1.5-1 demonstrates the dynamic page caching concept. Figure 6.1.5-2 provides the VHDL model.



**Figure 6.1.5-1 Dynamic Page Caching**

```

library IEEE;
use IEEE.Std_Logic_1164.ALL;
entity RAM is
  generic(DataWidth_g : positive := 16;
          AddrWidth_g : positive := 32);
  port(Address      : in  Std_Logic_Vector(AddrWidth_g - 1 downto 0);
        Data        : inout Std_Logic_Vector(DataWidth_g - 1 downto 0);
        CeF         : in  Std_Logic;
        RdWrF       : in  Std_Logic);
end RAM;

architecture RAMLINK_A of RAM is
begin
  Mem_Lbl: process(Address, Data, RdWrF, CeF)
    constant BlockSize_c : Positive := 12; -- memory block per page
                                              -- in bit width, 12 bits = 4 K words
    subtype DataWidth_Typ is Std_Logic_Vector(DataWidth_g - 1 downto 0);

```

```

type RamBlock_Typ is array (0 to 2 ** BlockSize_c - 1) of DataWidth_Typ;
subtype PageAddr_Typ is Integer range Address'left downto BlockSize_c;
type PageBlock_Typ;           -- incomplete type declaration
type PagePntr_Typ is access PageBlock_Typ;
type PageBlock_Typ is record
  Page    : Std_Logic_Vector(PageAddr_Typ);
  RAM     : RamBlock_Typ;      -- RAM Block
  NextP   : PagePntr_Typ;
end record;

function Vect2Int (Arg_c : in Std_Logic_Vector) return Integer is
  use IEEE.Numeric_Std.all;
begin
  return IEEE.Numeric_Std.TO_INTEGER(UNSIGNED(Arg_c));
end;

variable FoundAddr_v : Boolean;
variable DoneSearch_v : Boolean;
variable Head_v       : PagePntr_Typ;
variable Temp_v       : PagePntr_Typ;
variable New_v        : PagePntr_Typ;
variable AddrInt_v   : Natural;  -- converted address

begin
  if (CeF = '0') or (CeF = 'L') then          -- level 1
    FoundAddr_v := False;
    DoneSearch_v := False;
    if Head_v /= null then                    -- level 2
      -- Page exists
      Temp_v := Head_v;
      Search_Lbl: while not DoneSearch_v loop
        -- Check for availability of a block of data in a link
        if Temp_v.Page = Address(Address'left downto BlockSize_c) then -- lvl 3
          FoundAddr_v := True;
          DoneSearch_v:= True;
          if RdWrF'event and -- check for rising edge for a write -- level 4
            ((RdWrF = '1') or (RdWrF = 'H')) and
            ((RdWrF'last_value = '0') or (RdWrF'last_value = 'L')) then
            AddrInt_v := Vect2Int(Address(BlockSize_c - 1 downto 0));
            Temp_v.Page := Address(Address'left downto BlockSize_c);
            Temp_v.Ram(AddrInt_v) := Data;
          elsif ((RdWrF = '1') or (RdWrF = 'H')) then
            AddrInt_v := Vect2Int(Address(BlockSize_c - 1 downto 0));
            Data <= Temp_v.Ram(AddrInt_v);
          elsif RdWrF'event and -- check for falling edge for a write
            ((RdWrF = '0') or (RdWrF = 'L')) and
            ((RdWrF'last_value = '1') or (RdWrF'last_value = 'H')) then
            null;
          else
            assert False
              report "Error in RdWrF signal"
              severity Warning;
            end if;                                -- level 4
          elsif Temp_v.NextP = null then
            exit Search_Lbl;
          else
            Temp_v := Temp_v.NextP;
          end if;
        end loop;
      end if;
    end if;
  end if;
end;

```

```

-- header is null
elsif RdWrF'event and      -- check for rising edge for a write,
  ((RdWrF = '1') or (RdWrF = 'H')) and
  ((RdWrF'last_value = '0') or (RdWrF'last_value = 'L')) then
  -- a 1st write
  Head_v := new PageBlock_Typ;
  AddrInt_v := Vect2Int(Address(BlockSize_c - 1 downto 0));
  Head_v.Page := Address(Address'left downto BlockSize_c);
  Head_v.Ram(AddrInt_v) := Data;
  Head_v.NextP := null;
  FoundAddr_v := True;
end if;                                -- level 2

if not FoundAddr_v then                -- level 2B
  if RdWrF'event and      -- check for rising edge for a write -- level 3B
    ((RdWrF = '1') or (RdWrF = 'H')) and
    ((RdWrF'last_value = '0') or (RdWrF'last_value = 'L')) then
    New_v := new PageBlock_Typ;
    AddrInt_v := Vect2Int(Address(BlockSize_c - 1 downto 0));
    New_v.Page := Address(Address'left downto BlockSize_c);
    New_v.Ram(AddrInt_v) := Data;
    Temp_v.NextP := New_v;
  elsif ((RdWrF = '1') or (RdWrF = 'H')) then
    Data <= (others => 'X');

  elsif RdWrF'event and      -- check for falling edge for a write
    ((RdWrF = '0') or (RdWrF = 'L')) and
    ((RdWrF'last_value = '1') or (RdWrF'last_value = 'H')) then
    null;

  else
    assert False
      report "Error in RdWrF signal"
      severity Warning;
  end if;                                -- level 3B
  end if;                                -- level 2B

else -- Ce_F /= '0' | 'L'
  data <= (others => 'Z');               -- level 1
end if;
end process Mem_Lbl;
end RamLink_a;

```

**Figure 6.1.5-1 Dynamic Page Caching (models\ramlink.vhd)**

### 6.1.6 Disk paging with swap files

The disk paging method uses swap files (on disk) to store pages of memory, and a modeling RAM (similar to the model described in section 5.1.1) to hold the currently accessed page. The memory address range can be large, but the RAM model held by the simulator can be kept much smaller. When a new page is requested, the contents of the currently held page (in modeling RAM) is saved into a file. The requested page, corresponding to the new address, is retrieved from the disk into the modeling RAM. Note that unlike VHDL'93, VHDL'87 does not allow file closing and reopening. As discussed in sections 4.4 and 4.5, procedures with local file declarations (and strings passed as actual parameters for the file names) can be used to emulate this behavior. This modeling style is not recommended because disk paging is not efficient when often swapping is required. The modeling of such an approach is left as an exercise.

### 6.1.7 C Language Interface

This approach may be efficient; however it is not recommended as a general solution because it is non-portable.

### 6.1.8 RAM Testbench and Configurations

Figure 6.1.8-1 provides a testbench for the *RAM* models. Figures 6.1.8-2 through 6.1.8-3 define testbench configurations for the *RamFix\_a*, *RamPage\_a*, and *RamLink\_a* architectures.

```

library IEEE;
use IEEE.Std_Lock_1164.all;
entity RamTB is
end RamTb;

architecture RamTb_a of RamTB is
  constant DataWidth_c : Positive := 16;
  constant AddrWidth_c : Positive := 32;
  component RAM
    generic(DataWidth_g : Positive := 16;
            AddrWidth_g : Positive := 32);
    port(Address      : in  Std_Lock_Vector(AddrWidth_g -1 downto 0);
          Data        : inout Std_Lock_Vector(DataWidth_g -1 downto 0);
          CeF         : in  Std_Lock;
          RdWrF       : in  Std_Lock);
  end component;
  signal Address      : Std_Lock_Vector(AddrWidth_c -1 downto 0);
  signal Data        : Std_Lock_Vector(DataWidth_c -1 downto 0);
  signal CeF         : Std_Lock;
  signal RdWrF       : Std_Lock;
begin
  U1_RAM: RAM
    generic map
      (DataWidth_g => DataWidth_c,
       AddrWidth_g => AddrWidth_c)
    port map
      (Address      => Address,
       Data        => Data,
       CeF         => CeF,
       RdWrF       => RdWrF);

  DriveRam_Lbl: process
  begin
    -- WRITE
    Address <= "00001010100000010101010000000101" after 0 ns;
    Data    <= "0001010100000010" after 0 ns;
    CeF    <= '0' after 0 ns,
           '1' after 55 ns;
    RdWrF <= '0' after 0 ns,
           '1' after 50 ns;
    wait for 100 ns;
    -- READ
    Address <= "00001010100000010101010000000101" after 0 ns;
    Data    <= "ZZZZZZZZZZZZZZZ" after 0 ns;
    CeF    <= '0' after 0 ns,
           '1' after 55 ns;
  end process;
end;

```

```

RdWrF    <= '1' after 0 ns,
          '1' after 50 ns;
wait for 100 ns;

-- WRITE
Address <= "10001010100000010101010000000101" after 0 ns;
Data     <= "0001010100000010" after 0 ns;
CeF      <= '0' after 0 ns,
          '1' after 55 ns;
RdWrF    <= '0' after 0 ns,
          '1' after 50 ns;
wait for 100 ns;

-- READ
Address <= "10001010100000010101010000000101" after 0 ns;
Data     <= "ZZZZZZZZZZZZZZZ" after 0 ns;
CeF      <= '0' after 0 ns,
          '1' after 55 ns;
RdWrF    <= '1' after 0 ns,
          '1' after 50 ns;
wait for 1000 ns;
end process DriveRam_Lbl;
end RamTb_a;

```

**Figure 6.1.8-1 Memory Testbench (Models\ram\_tb.vhd)**

```

configuration RamFix_cfg of RamTB is
  for RamTB_a
    for U1_Ram: RAM use
      entity Work.RAM(RamFix_a)
        generic map(DataWidth_g => 16,
                    AddrWidth_g => 8);
    end for;
  end for;
end RamFix_Cfg;

```

**Figure 6.1.8-2 RamFix Configuration (models\ramfix\_c.vhd)**

```

configuration RamPage_cfg of RamTB is
  for RamTB_a
    for U1_Ram: RAM use
      entity Work.RAM(RamPage_a)
        generic map(DataWidth_g => 16,
                    AddrWidth_g => 12);
    end for;
  end for;
end RamPage_Cfg;

```

**Figure 6.1.8-2 RamPage Configuration (models\rampag\_c.vhd)**

```

configuration RamLink_cfg of RamTB is
  for RamTB_a
    for U1_Ram: RAM use
      entity Work.RAM(RamLink_a)
      generic map(DataWidth_g => 16,
                   AddrWidth_g => 32);
    end for;
  end for;
end RamLink_Cfg;

```

**Figure 6.1.8-2 RamLink Configuration (models\ramlink\_c.vhd)**

## 6.2 ZERO OHM RESISTOR (WIRE, BRIDGE) MODEL

**Q** How can a jumper wire or a bridge be modeled? The following code fails to work.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity ZeroOhm is
  port
    (A : inout Std_Logic;
     B : inout Std_Logic
    );
end ZeroOhm;

architecture ZeroOhm_a of ZeroOhm is
begin
  ABC0_Lbl: process
  begin
    wait on A'transaction, B'transaction;
    A <= B;
    B <= A;
  end process ABC0_Lbl;
end ZeroOhm_a;

```

**A** The proposed model fails because it does not detect the values driven on its ports when a transaction on either port (*A* or *B*) occurs. For example, if the model drives a '1' onto port *B*, and '0' is sourced on the signal associated with port *B*, then that signal is resolved to an 'X'. The model detects this transaction, and drives an 'X' on port *A*. If a '1' is later sourced on the signal associated with port *B*, then the model will drive an 'X' (the value of *A* port) on port *B*. Therefore, this model will latch the value of 'X' on its interface ports.

The solution to this model is similar to the proposed model. The entity consists of two bi-directional pins (*A* and *B*) of type *Std\_Logic*. The architecture is shown graphically in figure 6.2-1, and in VHDL in figure 6.2-2. The model utilizes the concept of a "BREAK BEFORE MAKE." Upon sensing a transaction on either port *A* or *B*, the model creates a "BREAK" by driving a 'Z' on both ports *A* and *B*. As a result of the *Std\_Logic* resolution function, the internal resistor drivers are automatically discounted, and the *A* and *B* ports resolve to the values of the active drivers that are connected on those pins. The *wait for 0 ns* statement insures the computation of the values of ports *A* and *B* when the model in the

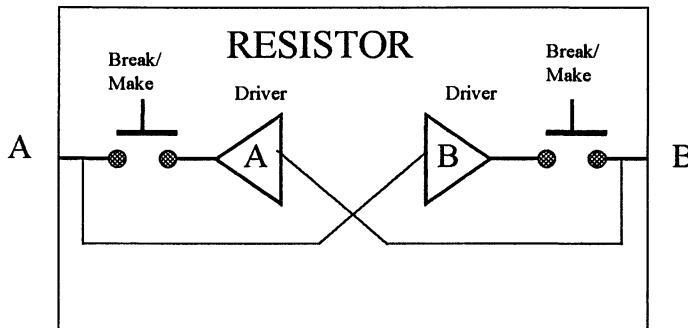
"BREAK" condition.

The "MAKE" occurs with the swap signal assignment statements:

```
A <= B;
B <= A;
```

The model uses the *wait ... until* clause as shown below to insure a single activation of the process during any simulation time. The *ThenTime\_v* is meant to discard the transactions generated by the "MAKE" section.

```
wait on A'transaction, B'transaction until ThenTime_v /= now;
```



**Figure 6.2-1 Graphical View of Resistor Model**

```
library IEEE;
use IEEE.Std_Lock_1164.all;
entity ZeroOhm1 is
port
(A : inout Std_Lock;
B : inout Std_Lock
);
end ZeroOhm1;

architecture ZeroOhm1_a of ZeroOhm1 is
begin
ABC0_Lbl: process
variable ThenTime_v : time;
begin
wait on A'transaction, B'transaction until ThenTime_v /= now;
-- Break
ThenTime_v := now;
A <= 'Z';
B <= 'Z';
wait for 0 ns;

-- Make
A <= B;
B <= A;
end process ABC0_Lbl;
end ZeroOhm1_a;
```

**Figure 6.2-2 Resistor Model (models\zohm0\_ea.vhd)**

A variation to the zero ohm resistor model (shown in figure 6.2-3) makes use of the '*quiet*' attribute to insure the break condition. The *A*'*quiet* and *B*'*quiet* are implicit signals, and thus cost a little more in terms of simulation efficiency. In addition, this approach requires more delta times than the solution proposed in figure 6.2-2. However, it is a viable solution that provides another dimension in the use of VHDL. Both resistor models have the same restrictions described in section 6.2.1.

```
architecture ZeroOhm1_a of ZeroOhm1 is
begin
  ABC0_Lbl: process
  begin
    wait on A'transaction, B'transaction;

    -- Break
    A <= 'Z';
    wait until A'quiet(0 ns); -- The (0 ns) is optional because
                               -- it is the default
    B <= 'Z';
    wait until B'quiet(0 ns);

    -- Make
    A <= B;
    wait until A'quiet(0 ns);

    B <= A;
    wait until B'quiet(0 ns);
  end process ABC0_Lbl;
end ZeroOhm1_a;
```

**Figure 6.2-3 Resistor Model with 'quiet Attribute (models\z0quiet.vhd)**

The model can be generalized for a zero ohm resistor bank with a width defined by a generic. This is shown in figure 6.2-4.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity ZeroOhm is
  generic (Width_g      : Positive := 32);
  port
    (A : inout Std_Logic_Vector(Width_g - 1 downto 0)
     := (others => 'Z');
     B : inout Std_Logic_Vector(Width_g - 1 downto 0)
     := (others => 'Z'));
end ZeroOhm;
```

```

architecture ZeroOhm_a of ZeroOhm is
begin
  Px_Lbl: for I in (Width_g - 1) downto 0 generate
    ABC0_Lbl: process
      variable ThenTime_v : time;
    begin
      wait on A(I)'transaction, B(I)'transaction
          until ThenTime_v /= now;
      -- Break
      ThenTime_v := now;
      A(I) <= 'Z';
      B(I) <= 'Z';
      wait for 0 ns;

      -- Make
      A(I) <= B(I);
      B(I) <= A(I);

    end process ABC0_Lbl;
  end generate Px_Lbl;
end ZeroOhm_a;

```

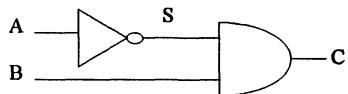
**Figure 6.2-4 Resistor Bank Model (models\zohm\_ea.vhd)**

The model utilizes the *generate* constructs to produce code that is more efficient since it generates one process for every bit of the resistor bank. Note that the LRM specifies that if a composite (array or record) signal is resolved at the composite level, then a change to one element causes an activity on the whole composite (see section 2.1, drivers of subelements of composite types). However, *Std\_Logic\_Vector* type is defined as an unconstrained array of *Std\_Logic*, and thus each subelement is resolved independently, so they act as separate signals (see section 3.1.3 for a discussion of atomicity).

When making port associations to the *ZeroOhm* component, avoid using the subelement association method because it is not portable. Instead, use as many resistor instantiations as necessary, with each resistor being either a bit device or an array equal to the size of the busses that are in use. For example, if an architecture includes signals *BusA32*, *BusB16*, *BusC1*, it is recommended to use three resistors of size 32, 16 and 1. Port mapping is to be performed on a bus basis (e.g., port map (*A* => *BusA32*, *B* => *BusB16*, *C* => *BusC1*)). See the testbench model of the "Error Injector" for an example of component instantiations of the error injector component. This is similar in construct to the resistor model.

### 6.2.1 Zero Ohm Resistor Model Restrictions

Signals asserted on the ports of the error injector should not have transactions occurring in multiple delta times as the model is sensitive to transactions on port A, B, **ONLY ONCE** during a simulation time. Once fired, a process will not refire when there are multiple transactions occurring in delta times. This condition can occur in gate level simulations with ZERO delays because transactions can happen in multiple delta times. For example:



If inputs A and B both transition at time X from a “00” to a “11”, then at time “X + 1 delta” signal S will transition from a ‘1’ to a ‘0’. Output C will also transition from a ‘0’ to a ‘1’. At time “X + 2 deltas” output C will transition from a ‘1’ to ‘0’. Thus, two transitions occur on output C in multiple delta times.

Another limitation to the resistor model is that data on ports A and B CANNOT be of value ‘-’. This is because the model asserts the value ‘Z’ on those ports to determine the driving value on ports A and B. The *Std\_Logic\_1164* package resolves a ‘-’ with anything but a ‘U’ to an ‘X’.

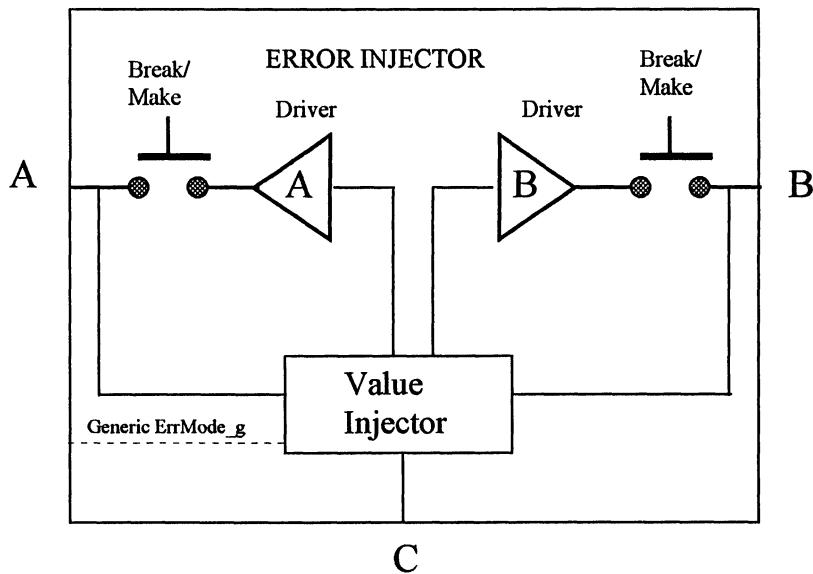
### 6.2.2 Architecture Variation Method for Resistor

If the application is to simulate an ASIC having two ports that could be shorted in a board architecture, then it is possible to use two board level architectures, one with the short (bridge ON), and the other with the pins opens. For a short, port associations can be made to the signal that connects the two pins. For an open, the port associations can be made to the *open*. When defining the port disconnection (i.e. the *open*), it is important to abide by the port association rules discussed in section 1.4.3. Configuration declarations can be used to define the requested configuration (bridge ON/OFF).

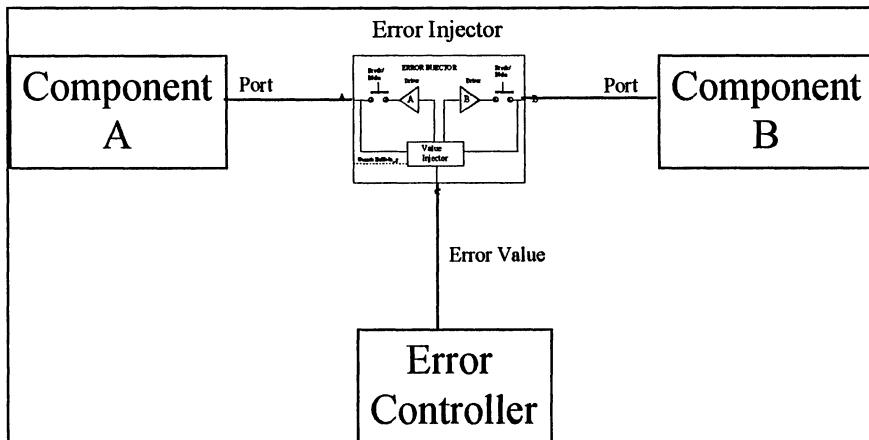
## 6.3 ERROR INJECTOR MODEL

**Q** How can a model be created to force errors onto a signal? This error injector model must be controlled in a dynamic manner, with the error enabled or disabled. This model is useful in the verification of multiple ASIC designs, where an error (e.g. parity error, data, address) must be injected to verify the ASICs responses to such error conditions.

**A** A three port (*A*, *B*, *C*) error injector component could be used to act as either a zero ohm resistor (see section 6.2) between ports *A* and *B*, or as a forced value injected on either port *A* or *B*. The control of the injected errors into these two ports (*A*, *B*) is determined by the value asserted on port *C*. Figure 6.3-1 demonstrates the architecture of this model. Figure 6.3-2 demonstrates its application in a subsystem. This component effectively emulates a stuck at fault, thus emulating an open, a short to Vcc, or a short to ground. The values asserted to ports *A* and *B* are of type *Std\_Logic*, and are resolved. The component can be dynamically controlled to operate in either normal mode (*A* and *B* become a jumper wire passing data in either direction), or in fault injection mode (error value is asserted onto the desired ports (*A* or *B* or both)).



**Figure 6.3-1 Error Injector Architecture**



**Figure 6.3-2 Error Injector Utilization**

The applications of this component include:

- Normal operation of a jumper wire (data flowing in both directions),
- Stuck at fault where the value of the fault is asserted on port C. The timing of the fault is controlled by the transactions of port C. The mode of the stuck at fault is determined by a generic.
- Testing of setup time by forcing signals to forced values, such as 'X', until setup time.
- Dynamic disconnect of a component pin from its normal connection for testing or for fault isolation.

The error injector component consists of three ports:

- **Port A:** One side of the pass-through switch
- **Port B:** The other side of the pass-through switch
- **Port C:** The dynamic control port for the pass-through switch. This control defines operation of the switch as either a straight pass-through (i.e., zero ohm resistor) when the value is '**-**', or a forced value (when other than '**-**'). The forced values, to be asserted as faults, can be '**U**' '**X**' '**0**' '**1**' '**Z**' '**W**' '**L**' '**H**'.

The model is sensitive to transactions on all ports. Once a transaction is detected, all other transactions are ignored for that simulation time (i.e., further transactions in that delta time are ignored).

Two models are provided, one with the interfaces of type *Std\_Logic\_Vector*, and the other of type *Std\_Logic*. For the vector model, the width of the pass-through switch is defined by the generic *width\_g*. The pass-through control is defined on a per bit basis (i.e., one process per bit).

In both models, the mode of the error injection is defined by the generic *InjMode\_g*; this can take one of the values defined in table 6.3.

The model limitations and restrictions for the error injector are the same as those described in section 6.2, except that the model is also sensitive to transactions on port *C*.

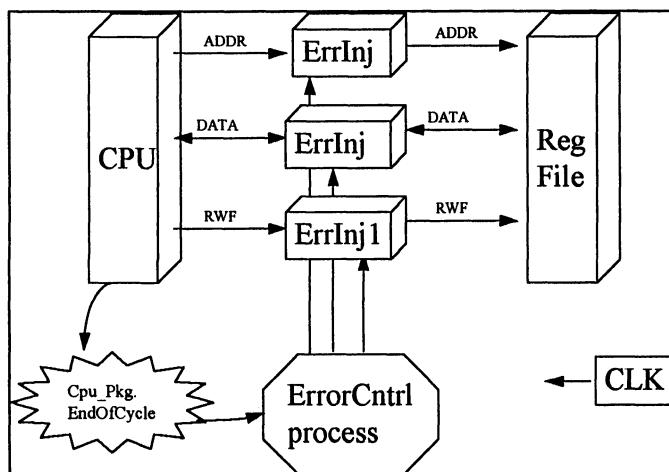
A possible variation to this model is the addition of a fourth port (*D*) that is of type *InjMode\_Typ* or of type *InjModeAry\_Typ* to dynamically identify the fault mode.

Figure 6.3-3 is a top level view of the testbench. It incorporates the CPU and Register File BFM's, the Error Injector components, the clock generator, and the error control process. This process is synchronized with a global signal that is asserted by the CPU model. Figure 6.3-4 represents the error injector model. Figure 6.3-5 and 6.3-6 are models of a simple CPU and Register file used in the testbench. The testbench is defined in figure 6.3-7. A set of configuration declarations is included in figure 6.3-8. The compilation order for these files is as shown below. The files must be compiled in library *Work*.

1. errinj.vhd
2. ej\_cpu.vhd
3. ej\_reg.vhd
4. ej\_tb.vhd
5. ej\_cfg.vhd

**Table 6.3 Error Injector Mode and Operations**

MODE	OPERATION	COMMENTS
NoError	Control C has no influence on the switch.	Ports A and B always operate as a pass-through between A(i) and B(i).
StuckA	When C(i) = '-'	Ports A and B always operate as a pass-through between A(i) and B(i).
	When C(i) /= '-'	Value of C(i) is driven on "A" port of component. 'Z' is driven on "B" port of component.
StuckB	When C(i) = '-'	Ports A and B always operate as a pass-through between A(i) and B(i).
	When C(i) /= '-'	Value of C(i) is driven on "B" port of component. 'Z' is driven on "A" port of component
StuckAB	When C(i) = '-'	Ports A and B always operate as a pass-through between A(i) and B(i).
	When C(i) /= '-'	Value of C(i) is driven on "B" port of component. Value of C(i) is driven on "A" port of component
DriverAB	When C(i) = '-'	Ports A and B always operate as a pass-through between A(i) and B(i).
	When C(i) /= '-'	if one side has a strong drive, and the other side has a weak drive then value of C(i) is driven on the weak drive port, and 'Z' is driven on the strong drive port. Else, Switch operate as a pass-through between A(i) and B(i) (i.e. no error).

**Figure 6.3-3 Error Injector Testbench Architecture**

```

-- Package Name : ErrInj_Pkg
-- Library      : "Work"
-- Purpose       : Declaration of enumeration type for
--                  : use by error injector component
-----
library IEEE;
use IEEE.Std_Logic_1164.all;

package ErrInj_Pkg is
  type InjMode_Typ is
    (NoError,   -- A and B is a jumper wire, data in either direction
     StuckA,    -- If C(i) /= '-' then A <= C(i); B <= 'Z' else **
     StuckB,    -- If C(i) /= '-' then B <= C(i); A <= 'Z' else **
     StuckAB,   -- If C(i) /= '-' then A <= C(i); B <= C(i) else **
     DriverAB); -- If C(i) /= '-' then
                  -- if A is strong driver and B weak driver then
                  --     A <= 'Z'; B <= C(i)
                  -- elsif B is strong driver and A weak driver then
                  --     B <= 'Z'; A <= C(i)
                  -- else A and B is a jumper wire
                  --
                  -- jumper wire
                  -- ** if C(i) = '-' then A <= B; B <= A
    -- The following type can be used to define the type of a
    -- fourth port to the model to
    -- identify the error injection mode. This type is not used in this model.
    type InjModeAry_Typ is array(Integer range <>) of InjMode_Typ;
end ErrInj_Pkg;
-----

-- Entity        : ErrInj
-- Purpose       : Provides interface of a component intended
--                  : to be placed between signals
--                  : to dynamically control faults on signals.
-----
library IEEE;
use IEEE.Std_Logic_1164.all;

library Work;
use Work.ErrInj_Pkg.all;
use Work.ErrInj_Pkg;
entity ErrInj is
  generic (Width_g      : Positive := 32;
           InjMode_g    : ErrInj_Pkg.InjMode_Typ := NoError);
  port
    (A : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     B : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     C : in    Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z'));
end ErrInj;
-----

-- Architecture : ErrInj_a
-- Purpose      : Implements the error injection architecture.
--                  : This model is optimized for speed by minimizing the
--                  : number of transactions by generating one process
--                  : for each element and mode of interest.
-----
```

```

architecture ErrInj_a of ErrInj is
  type TwoDbool_Typ is array(Std_Logic, Std_Logic) of Boolean;
  constant T : Boolean := True;
  constant F : Boolean := False;

  -- True if ROW is Strong and column is weak.
  constant StrongWeak_c : TwoDbool_Typ := (
    -- accessed as (Row, Column)
    -- Side2 signal (column)
    --   U   X   0   1   Z   W   L   H   -
    -----
    ( F , F , F , F , T , T , T , T , T ), --U Side 1 signal
    ( F , F , F , F , T , T , T , T , T ), --X (Row)
    ( F , F , F , F , T , T , T , T , T ), --0
    ( F , F , F , F , T , T , T , T , T ), --1

    ( F , F , F , F , F , F , F , F , F ), --Z
    ( F , F , F , F , F , F , F , F , F ), --W
    ( F , F , F , F , F , F , F , F , F ), --L
    ( F , F , F , F , F , F , F , F , F ), --H
    ( F , F , F , F , F , F , F , F , F ) ---)
  );

begin
  Px_Lbl: for I in (Width_g - 1) downto 0 generate
    NoError_Lbl: if (InjMode_g = NoError) generate
      ABC0_Lbl: process
        variable ThenTime_v : time;
      begin
        wait on A(I)'transaction, B(I)'transaction until ThenTime_v /= now;
        -- Break
        ThenTime_v := now;
        A(I) <= 'Z';
        B(I) <= 'Z';
        wait for 0 ns;

        -- Make
        A(I) <= B(I);
        B(I) <= A(I);
      end process ABC0_Lbl;
    end generate NoError_Lbl;

    StuckA_Lbl: if (InjMode_g = StuckA) generate
      ABC1_Lbl: process
        variable ThenTime_v : time;
      begin
        wait on A(I)'transaction, B(I)'transaction, C(I)'transaction
          until ThenTime_v /= now;
        -- Break
        ThenTime_v := now;
        A(I) <= 'Z';
        B(I) <= 'Z';
        wait for 0 ns;
        -- Make
        if C(I) /= '-' then
          A(I) <= C(I); -- A stuck at fault
          B(I) <= 'Z'; -- B is unaffected
          -- (no driver from error injector)
        else
          A(I) <= B(I);
          B(I) <= A(I);
        end if;
      end process ABC1_Lbl;
    end generate StuckA_Lbl;
end;

```

```

StuckB_Lbl: if (InjMode_g = StuckB) generate
  ABC2_Lbl: process
    variable ThenTime_v : time;
  begin
    wait on A(I)'transaction, B(I)'transaction, C(I)'transaction
      until ThenTime_v /= now;
    -- Break
    ThenTime_v := now;
    A(I) <= 'Z';
    B(I) <= 'Z';
    wait for 0 ns;

    -- Make
    if C(I) /= '-' then
      B(I) <= C(I);    -- B stuck at fault
      A(I) <= 'Z';    -- A is unaffected
      -- (no driver from error injector)
    else
      A(I) <= B(I);
      B(I) <= A(I);
    end if;
  end process ABC2_Lbl;
end generate StuckB_Lbl;

StuckAB_Lbl: if (InjMode_g = StuckAB) generate
  ABC3_Lbl: process
    variable ThenTime_v : time;
  begin
    wait on A(I)'transaction, B(I)'transaction, C(I)'transaction
      until ThenTime_v /= now;
    -- Break
    ThenTime_v := now;
    A(I) <= 'Z';
    B(I) <= 'Z';
    wait for 0 ns;

    -- Make
    if C(I) /= '-' then
      A(I) <= C(I);    -- A stuck at fault
      B(I) <= C(I);    -- B stuck at fault

    else
      A(I) <= B(I);
      B(I) <= A(I);
    end if;
  end process ABC3_Lbl;
end generate StuckAB_Lbl;

DriverAB_Lbl: if (InjMode_g = DriverAB) generate
  ABC4_Lbl: process
    variable ThenTime_v : time;
  begin
    wait on A(I)'transaction, B(I)'transaction, C(I)'transaction
      until ThenTime_v /= now;
    -- Break
    ThenTime_v := now;
    A(I) <= 'Z';
    B(I) <= 'Z';
    wait for 0 ns;

    -- Make
    if C(I) /= '-' then
      -- Test if A is strong, and B is weak driver
      if StrongWeak_c(A(I), B(I)) then
        B(I) <= C(I);    -- B gets at fault
        A(I) <= 'Z';    -- A is unaffected
        -- (no driver from error injector)
      end if;
    end if;
  end process ABC4_Lbl;
end generate DriverAB_Lbl;

```

```

-- Test if B is strong, and A is weak driver
elsif StrongWeak_c(B(I), A(I)) then
    A(I) <= C(I);      -- A gets at fault
    B(I) <= 'Z';       -- B is unaffected
                        -- (no driver from error injector)
-- Pass data with no fault injection
-- because A and B are either both strong
-- or both weak drivers
else
    A(I) <= B(I);
    B(I) <= A(I);
end if;
else                                -- C(I) = '-'
    A(I) <= B(I);
    B(I) <= A(I);
end if;
end process ABC4_Lbl;
end generate DriverAB_Lbl;
end generate Px_Lbl;
end ErrInj_a;

-----
----- Error Injector - 1 bit -----
-----
-- Entity      : ErrInj1
-----
library IEEE;
use IEEE.Std_Logic_1164.all;

library Work;
use Work.ErrInj_Pkg.all;
use Work.ErrInj_Pkg;

entity ErrInj1 is
    generic (InjMode_g      : ErrInj_Pkg.InjMode_Typ := NoError);
    port
        (A : inout Std_Logic := 'Z';
         B : inout Std_Logic := 'Z';
         C : in    Std_Logic := 'Z');
end ErrInj1;

-- Architecture : ErrInj1_a
-- Purpose      : Implements the error injection architecture.
-----
architecture ErrInj1_a of ErrInj1 is
    -- See file for details. Model is very similar to ErrInj_a architecture.
end ErrInj1_a;

```

**Figure 6.3-4 Error Injector Model (models\errinj.vhd)**

```

package Cpu_Pkg is
    signal EndOfCycle_s : Boolean := False; ← Signal Assigned by CPU model to synchronize error injection process in the testbench model.
end Cpu_Pkg;

library IEEE;
use IEEE.Std_Lock_1164.all;
library Work;
use Work.Cpu_Pkg.all; use Work.Cpu_Pkg;
entity Cpu is
    generic(Delay_g : time := 10 ns);
    port
        (ADDR : out Std_Lock_Vector(1 downto 0)
         := (others => '1'); -- 2 bit address
        DATA : inout Std_Lock_Vector(7 downto 0)
         := (others => 'Z'); -- 8 bit data
        RWF : out Std_Lock := '1'; -- Read = '1', Write = '0'
        Clk : in Std_Lock := '1'); -- For timing synchronization
end Cpu;

Architecture Cpu_a of Cpu is
    subtype Std2Bits_Typ is Std_Lock_Vector(1 downto 0);
    subtype Intgr03_Typ is Integer range 0 to 3;
    type TwoD_Typ is array(Intgr03_Typ) of Std2Bits_Typ;
    constant Addr_c : TwoD_Typ := ("00", "01", "10", "11");

begin
    Transaction_Lbl: process
        variable NextAddr_v : Intgr03_Typ := 0;
        variable Data_v : Std_Lock_Vector(7 downto 0) := "10110010";
        constant Delta_c : Time := Delay_g / 5;

        begin
            wait until Clk'event and Clk = '1';
            if NextAddr_v = 0 then -- Set up for synchronization of testbench
                Cpu_Pkg.EndOfCycle_s <= True, False after 1 ns;
            end if;

            -- Do a WRITE
            ADDR <= Addr_c(NextAddr_v) after Delay_g; -- Address
            RWF <= '0' after Delay_g - Delta_c; -- Control
            Data <= Data_v after 2 * Delta_c; -- Data

            wait until Clk'event and Clk = '1';
            -- Do a READ
            ADDR <= Addr_c(NextAddr_v) after Delay_g; -- Address
            RWF <= '1' after Delay_g - Delta_c; -- Control
            Data <= (others => 'Z') after 2 * Delta_c; -- read

            -- Setup for next Write/Read
            NextAddr_v := (NextAddr_v + 1) mod (Intgr03_Typ'High + 1);
            Data_v := Data_v(6 downto 0) & Data_v(7); -- Rotate left logical
        end process Transaction_Lbl;
    end Cpu_a;

```

**Figure 6.3-5 Simple CPU Bus Functional Model for use in the Error Injector Testbench (Models\ej\_cpu.vhd)**

```

library IEEE;
use IEEE.Std_Lock_1164.all;
entity Reg is
    generic(Delay_g : Time := 14 ns);
    port
        (ADDR : in Std_Lock_Vector(1 downto 0); -- 2 bit address
        DATA : inout Std_Lock_Vector(7 downto 0) := (others => 'Z');
        RWF : in Std_Lock; -- Read = '1', Write = '0'
        Clk : in Std_Lock); -- For timing synchronization
end Reg;

```

```

architecture Reg_a of Reg is
  subtype Std8Bits_Typ is Std_Logic_Vector(7 downto 0);
  subtype Intgr03_Typ is Integer range 0 to 3;
  type TwoD_Typ is array(Intgr03_Typ) of Std8Bits_Typ;
begin
  Reg_Lbl: process(ADDR, DATA, RWF, CLK)
    variable RegFile_v : TwoD_Typ;
    constant Delta_c :Time := Delay_g / 5;
    variable Addr_v : Std_Logic_Vector(1 downto 0);
  begin
    Addr_v := To_X01(ADDR);
    if (RWF = '0' or RWF = 'L') and
       not (CLK'event and CLK = '1') then
      DATA <= (others => 'Z');
    elsif (RWF = '0' or RWF = 'L') and
          (Clk'event and CLK = '1') then
      case ADDR is
        when "00" =>
          RegFile_v(0) := To_X01(DATA); -- from Std_logic_1164 package
        when "01" =>
          RegFile_v(1) := To_X01(DATA);
        when "10" =>
          RegFile_v(2) := To_X01(DATA);
        when "11" =>
          RegFile_v(3) := To_X01(DATA);
        when others => null;
      end case;
    elsif (RWF = '1' or RWF = 'H') then -- Read
      case ADDR_v is
        when "00" =>
          DATA <= RegFile_v(0) after 3 * Delta_c;
        when "01" =>
          DATA <= RegFile_v(1) after 3 * Delta_c;
        when "10" =>
          DATA <= RegFile_v(2) after 3 * Delta_c;
        when "11" =>
          DATA <= RegFile_v(3) after 3 * Delta_c;
        when others =>
          DATA <= (others => 'X') after 3 * Delta_c;
      end case;
    else
      null;
    end if;
  end process Reg_Lbl;
end Reg_a;

```

**Figure 6.3-6 Register file BFM for use in the Error Injector Testbench  
(Models\ej\_reg.vhd)**

```

library IEEE;
use IEEE.Std_Logic_1164.all;

library Work;
use Work.ErrInj_Pkg.all;
use Work.ErrInj_Pkg; ← This declaration enables the shortening of the path name by
use Work.Cpu_Pkg.all; providing the package name without the library name.
use Work.Cpu_Pkg; ← Providing the package name enhances readability

entity TestBench is
end TestBench;

```

```

architecture TestBench_a of TestBench is
component Cpu
generic(Delay_g : time := 10 ns);
port
(ADDR : out Std_Logic_Vector(1 downto 0); -- 2 bit address
 DATA : inout Std_Logic_Vector(7 downto 0); -- 8 bit data
 RWF : out Std_Logic; -- Read = '1', Write = '0'
 Clk : in Std_Logic); -- For timing synchronization
end component;

component ErrInj
generic (Width_g      : Positive := 11;
         InjMode_g    : ErrInj_Pkg.InjMode_Typ := NoError);
port
(A : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
 B : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
 C : in  Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z'));
end component;

component ErrInj1
generic (InjMode_g    : ErrInj_Pkg.InjMode_Typ := NoError);
port
(A : inout Std_Logic := 'Z';
 B : inout Std_Logic := 'Z';
 C : in  Std_Logic := 'Z');
end component;

component Reg
generic(Delay_g : time := 14 ns);
port
(ADDR : in Std_Logic_Vector(1 downto 0); -- 2 bit address
 DATA : inout Std_Logic_Vector(7 downto 0); -- 8 bit data
 RWF : in Std_Logic; -- Read = '1', Write = '0'
 Clk : in Std_Logic); -- For timing synchronization
end component;

signal ADDRa_s : Std_Logic_Vector(1 downto 0) := (others => 'H');
signal ADDRb_s : Std_Logic_Vector(1 downto 0) := (others => 'H');
signal DATAa_s : Std_Logic_Vector(7 downto 0) := (others => 'H');
signal DATAb_s : Std_Logic_Vector(7 downto 0) := (others => 'H');
signal RWFa_s : Std_Logic := 'H'; -- Read = '1', Write = '0'
signal RWFb_s : Std_Logic := 'H'; -- Read = '1', Write = '0'
signal CNTRL_s : Std_Logic_Vector(10 downto 0) := (others => '-');
signal ADDRc_s : Std_Logic_Vector(1 downto 0) := (others => 'H');
signal DATAc_s : Std_Logic_Vector(7 downto 0) := (others => 'H');
signal RWFc_s : Std_Logic := 'H';
signal Clk     : Std_Logic := '1'; -- For timing synchronization

begin
Clk <= not Clk after 100 ns;

U1_Cpu: Cpu
generic map (Delay_g => 10 ns)
port map
(ADDR => ADDRa_s,
 DATA => DATAa_s,
 RWF  => RWFa_s,
 Clk  => CLK);

U1_ErrInj: ErrInj -- Address
generic map (Width_g      => 2,
             InjMode_g    => NoError)
port map
(A => ADDRa_s,
 B => ADDRb_s,
 C => ADDRc_s);

```

```

U2_ErrInj: ErrInj -- Data
  generic map (Width_g      => 8,
                InjMode_g    => NoError)
  port map
    (A => DATAa_s,
     B => DATAb_s,
     C => DATAc_s);

U3_ErrInj1: ErrInj1   -- RWF
  generic map (InjMode_g    => NoError)
  port map
    (A => RWFa_s,
     B => RWFB_s,
     C => RWFc_s);

U_Reg: Reg
  generic map (Delay_g => 14 ns)
  port map
    (ADDR => ADDRb_s,
     DATA  => DATAb_s,
     RWF   => RWFB_s,
     CLK   => CLK);

ErrorCntrl_Lbl: Process
  variable Count_v : integer := 0;
begin
  ----- Normal/Write fault -----
  assert False
  report "No Fault injection"
  severity Note;
  -- AB switch is pass though
  ADDRc_s <= "--"; -- transaction to fire the error injector
  DATAc_s <= "-----";
  RWFc_s <= '-';
  wait until Cpu_Pkg.EndOfCycle_s; -- ignore dummy 1st signal
  wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
  assert False
  report "Fault injection, WRF = 0"
  severity Note;
  -- Error insertion in WRF
  ADDRc_s <= "--";
  DATAc_s <= "-----";
  RWFc_s <= '0';

  ----- Normal/Read fault -----
  wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
  assert False
  report "No Fault injection"
  severity Note;
  -- AB switch is pass though
  ADDRc_s <= "--";
  DATAc_s <= "-----";
  RWFc_s <= '-';

  wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
  assert False
  report "Fault injection, WRF = 1"
  severity Note;
  -- Error insertion in WRF
  ADDRc_s <= "--";
  DATAc_s <= "-----";
  RWFc_s <= '1';

```

```
----- Normal/DATA fault -----
wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "No Fault injection"
  severity Note;
  -- AB switch is pass though
  ADDRc_s <= "--";
  DATAc_s <= "-----";
  RWFc_s <= '-';

wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "Fault injection, DATA = 01XLHUW-"
  severity Note;
  -- Error insertion in Data
  ADDRc_s <= "--";
  DATAc_s <= "01XLHUW-"; -- no error in bit 0
  RWFc_s <= '-';

----- Normal/Open DATA fault -----
wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "No Fault injection"
  severity Note;
  -- AB switch is pass though
  ADDRc_s <= "--";
  DATAc_s <= "-----";
  RWFc_s <= '-';

wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "Fault injection, DATA = HHHHHHHH"
  severity Note;
  -- Error insertion in Data
  ADDRc_s <= "--";
  DATAc_s <= "HHHHHHHH"; -- no error in bit 0
  RWFc_s <= '-';

----- Normal/Address fault -----
wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "No Fault injection"
  severity Note;
  -- AB switch is pass though
  ADDRc_s <= "--";
  DATAc_s <= "-----";
  RWFc_s <= '-';

wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "Fault injection, ADDRESS = 0-"
  severity Note;
  -- Error insertion in Address
  ADDRc_s <= "0-";
  DATAc_s <= "-----";
  RWFc_s <= '-';

----- Normal/OPEN fault -----
wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "No Fault injection"
  severity Note;
  -- AB switch is pass though
  ADDRc_s <= "--";
  DATAc_s <= "-----";
  RWFc_s <= '-';
```

```

wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "One side OPEN"
severity Note;
ADDRc_s <= "HH";
DATAc_s <= "HHHHHHHH";
RWFc_s <= 'H';

----- END of TEST -----
wait until Cpu_Pkg.EndOfCycle_s; -- All registers initialized in reg file
assert False
  report "end of test"
severity Note;
end process ErrorCntrl_Lbl;

end TestBench_a;

```

**Figure 6.3.7 Testbench for Error Injector Model (models\ej\_tb.vhd)**

```

library Work;
use Work.ErrInj_Pkg.all;

configuration StuckB_Cfg of TestBench is
  for TestBench_a
    for U1_ErrInj: ErrInj use
      entity Work.ErrInj(ErrInj_a)
        generic map (Width_g      => 2,
                     InjMode_g   => StuckB);
    end for;

    for U2_ErrInj: ErrInj use
      entity Work.ErrInj(ErrInj_a)
        generic map (Width_g      => 8,
                     InjMode_g   => StuckB);
    end for;

    for U3_ErrInj1: ErrInj1 use
      entity Work.ErrInj1(ErrInj1_a)
        generic map (InjMode_g   => StuckB);
    end for;
  end for;
end StuckB_Cfg;
--

library Work;
use Work.ErrInj_Pkg.all;

configuration StuckA_Cfg of TestBench is
  for TestBench_a
    for U1_ErrInj: ErrInj use
      entity Work.ErrInj(ErrInj_a)
        generic map (Width_g      => 2,
                     InjMode_g   => StuckA);
    end for;

    for U2_ErrInj: ErrInj use
      entity Work.ErrInj(ErrInj_a)
        generic map (Width_g      => 8,
                     InjMode_g   => StuckA);
    end for;

    for U3_ErrInj1: ErrInj1 use
      entity Work.ErrInj1(ErrInj1_a)
        generic map (InjMode_g   => StuckA);
    end for;
  end for;
end StuckA_Cfg;

```

```
--  
library Work;  
use Work.ErrInj_Pkg.all;  
  
configuration StuckAB_Cfg of TestBench is  
  for TestBench_a  
    for U1_ErrInj: ErrInj use  
      entity Work.ErrInj(ErrInj_a)  
        generic map (Width_g      => 2,  
                     InjMode_g    => StuckAB);  
    end for;  
  
    for U2_ErrInj: ErrInj use  
      entity Work.ErrInj(ErrInj_a)  
        generic map (Width_g      => 8,  
                     InjMode_g    => StuckAB);  
    end for;  
  
    for U3_ErrInj1: ErrInj1 use  
      entity Work.ErrInj1(ErrInj1_a)  
        generic map (InjMode_g    => StuckAB);  
    end for;  
  end for;  
end StuckAB_Cfg;  
--  
library Work;  
use Work.ErrInj_Pkg.all;  
  
configuration DriverAB_Cfg of TestBench is  
  for TestBench_a  
    for U1_ErrInj: ErrInj use  
      entity Work.ErrInj(ErrInj_a)  
        generic map (Width_g      => 2,  
                     InjMode_g    => DriverAB);  
    end for;  
  
    for U2_ErrInj: ErrInj use  
      entity Work.ErrInj(ErrInj_a)  
        generic map (Width_g      => 8,  
                     InjMode_g    => DriverAB);  
    end for;  
  
    for U3_ErrInj1: ErrInj1 use  
      entity Work.ErrInj1(ErrInj1_a)  
        generic map (InjMode_g    => DriverAB);  
    end for;  
  end for;  
end DriverAB_Cfg;  
*****
```

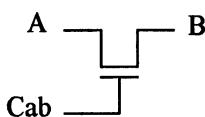
**figure 6.3-8 Configuration Declarations for Error Injector Test  
(models\ej\_cfg.vhd)**

## 6.4 TRANSFER GATE (SWITCH)

**Q** | How can a cascadable model of a transfer gate be designed? The operation of this device should be similar to the *Verilog* *tranifI* gate.

**A** | A transfer gate conducts when its control input is a *one*, and is nonconductive (or open) when the control input is a *zero*. Unlike *Verilog*, *VHDL* does not yet

provide such a primitive. However, a design approach similar to the modeling of the zero ohm resistor and error injector can be used. Figure 6.4-1 demonstrates the model of one non-cascadable transfer gate that operates on a set of busses. The entity/architecture



provide the definition of a three-port component (A, B, Cab). If Cab = '1' then A and B act as an ON switch (or zero ohm connection). Else, if Cab = '0', then A and B are assigned the value of 'Z'. The model is sensitive to transactions on all ports. Once a transaction is detected, all other transactions are ignored for that simulation time (i.e., further transactions in that delta time are ignored). The width of the pass-through switch is defined through the generic *width\_g*. The pass-through control and operation are defined on a per bit basis (i.e., one process per bit).

The model suffers from the same limitations as the zero ohm resistor model. Signals asserted on the ports of the switch should not have transactions occurring in multiple delta times because the model is sensitive to transactions on port A, B, Cab ONLY ONCE during a simulation time. Thus, once fired, a process will not refire if there are multiple transactions occurring in delta times. This condition may occur in gate level simulations with ZERO delays because transactions may occur in multiple delta times. The model makes use of a resolution table to compute the resolved value of A and B when the switch is in the ON state. That value is then assigned to ports A and B.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

entity Switch1 is
  generic (Width_g      : Positive := 32);
  port
    (A   : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     B   : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     Cab : in    Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z'));
end Switch1;

-----
-- Architecture  : Switch1_a
-- Purpose       : Implements the switch architecture.
--                 : This model is optimized for speed by minimizing the
--                 : number of transactions by generating one process
--                 : for each element of the width for the
--                 : mode of interest.
-----
```

```

architecture Switch1_a of Switch1 is
type StdLogic_Table is array(Std_uLogic, Std_uLogic) of Std_uLogic;

-----
-- resolution function, from 1164 package body
-----

constant Resolution_Table : Stdlogic_Table := (
-----
--   | U   X   0   1   Z   W   L   H   -   |   |
--   -----
--   ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
--   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
--   ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
--   ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
--   ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
--   ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
--   ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
--   ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
--   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
);

function Resolved ( S : Std_uLogic_Vector ) return Std_uLogic is
variable Result : Std_uLogic := 'Z'; -- weakest state default
begin
  -- the test for a single driver is essential otherwise the
  -- loop would return 'X' for a single driver of '-' and that
  -- would conflict with the value of a single driver unresolved
  -- signal.
  if (S'length = 1) then
    return S(S'low);
  else
    for I in S'range Loop
      Result := Resolution_Table(Result, S(I));
    end loop;
  end if;
  return Result;
end Resolved;

begin
Px_Lbl: for I in (Width_g - 1) downto 0 generate
  ABC1_Lbl: process
    variable ThenTime_v : time;
    variable ResolvedValue_v : Std_uLogic;
  begin
    wait on A(I)'transaction, B(I)'transaction, Cab(I)'transaction
    until ThenTime_v /= now;
    -- Break
    ThenTime_v := now;
    A(I) <= 'Z';
    B(I) <= 'Z';
    wait for 0 ns;

    if Cab(I) = '1' then
      -- Make. Must compute the resolved value
      ResolvedValue_v := Resolved((A(I), B(I)));
      A(I) <= ResolvedValue_v;
      B(I) <= ResolvedValue_v;

    else
      A(I) <= 'Z';
      B(I) <= 'Z';
    end if;
  end process ABC1_Lbl;
end generate Px_Lbl;
end Switch1_a;

```

**Figure 6.4-1 One Switch Model (models\switch1.vhd)**

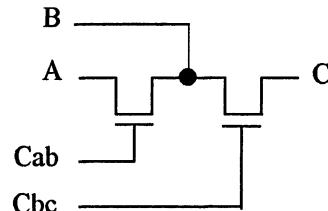
The same file (on disk) includes the model of a transfer gate with A, B and Cab of type *Std\_Logic*.

Since the switch model is not cascadable, the modeling of a cascaded two-switch model is provided in figure 6.4-2. That model is similar to the previous one, except that it is sensitive to all data and control ports.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity Switch2 is
  generic (Width_g      : Positive := 32);
  port
    (A   : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     B   : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     C   : inout Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     Cab : in   Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z');
     Cbc : in   Std_Logic_Vector(Width_g - 1 downto 0) := (others => 'Z')
    );
end Switch2;

architecture Switch2_a of Switch2 is
  type StdLogic_Table is array(Std_uLogic, Std_uLogic) of Std_uLogic;
  -----
  -- resolution function (from the 1164 package body)
  -----
  constant Resolution_Table : Stdlogic_Table := (
  -----
  --   | U   X   0   1   Z   W   L   H   -   |   |
  --   |-----|-----|-----|-----|-----|-----|-----|-----|-----|
  --   ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
  --   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
  --   ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
  --   ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
  --   ( 'U', 'X', '0', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
  --   ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
  --   ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
  --   ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
  --   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
  );
  function Resolved ( S : Std_uLogic_Vector ) return Std_uLogic is
    variable Result : Std_uLogic := 'Z'; -- weakest state default
  begin
    -- the test for a single driver is essential otherwise the
    -- loop would return 'X' for a single driver of '-' and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    if (S'length = 1) then
      return S(S'low);
    else
      for I in S'range Loop
        Result := Resolution_Table(Result, S(I));
      end loop;
    end if;
    return Result;
  end Resolved;
  subtype Std2_Typ is Std_Logic_Vector(0 to 1);
```



```

begin
Px_Lbl: for I in (Width_g - 1) downto 0 generate
    ABC1_Lbl: process
        variable ThenTime_v : time;
        variable ResolvedValue_v : Std_uLogic;
    begin
        wait on A(I)'transaction, B(I)'transaction, C(I)'transaction,
            Cab(I)'transaction, Cbc(I)'transaction
        until ThenTime_v /= now;
        -- Break
        ThenTime_v := now;
        A(I) <= 'Z';
        B(I) <= 'Z';
        C(I) <= 'Z';
        wait for 0 ns;

        case Std2_Typ'(Cab(I), Cbc(I)) is
            when "00" => -- both off
                A(I) <= 'Z'; -- redundant, already assigned a 'Z'
                B(I) <= 'Z'; -- redundant

            when "01" => -- Cab is off, Cbc is on
                A(I) <= 'Z'; -- redundant
                -- Make. Must compute the resolved value
                ResolvedValue_v := Resolved((B(I), C(I)));
                B(I) <= ResolvedValue_v;
                C(I) <= ResolvedValue_v;

            when "10" => -- Cab is on, Cbc is off
                C(I) <= 'Z'; -- redundant
                -- Make. Must compute the resolved value
                ResolvedValue_v := Resolved((A(I), B(I)));
                A(I) <= ResolvedValue_v;
                B(I) <= ResolvedValue_v;

            when "11" => -- Both on
                -- Make. Must compute the resolved value
                ResolvedValue_v := Resolved((A(I), B(I), C(I)));
                A(I) <= ResolvedValue_v;
                B(I) <= ResolvedValue_v;
                C(I) <= ResolvedValue_v;

            when others =>
                assert False
                    report "Cab and Cbc are not 0 or 1, Switches are OFF"
                    severity Note;

        end case;
    end process ABC1_Lbl;
end generate Px_Lbl;
end Switch2_a;

```

**Figure 6.4-2 Two Switches in Series Model (models\switch2.vhd)**

## **7. SYNTHESIS**

---

---

The majority of the examples and textual elaborations discussed in this section originated with Joseph Pick's excellent tutorial, *VHDL Synthesis Techniques and Recommendations*[<sup>11</sup>], presented at the 1996 Synopsys Users Group Conference.

*Synthesis is a translation process from an abstract description of a hardware device into a specific gate-level implementation of an optimized technology. Synthesis may be done:*

- *Manually via schematic entry.*
- *Automatically via design automation tools that use a Hardware Description Language (HDL) and an input medium to generate constraint driven gate configurations.*

*The HDL used to model targeted device is technology independent. During synthesis, appropriate cells are selected from the user specified technology library. The HDL coding styles is a very important driving force for the automatic synthesis algorithms and will play a key role in the final hardware configuration that is built.*

In this book, the HDL is VHDL. Though VHDL is a key element of a synthesis activity, it is only part of the solution. Other contributors include:

- *Target library design rules*
- *Hardware timing goals*
- *Hardware area goals*
- *Environmental constraints*
- *Proprietary Graphics User Interface (GUI) and command language features to ensure that the synthesized hardware will best satisfy all of its intended real-world design constraints and goals.*

## 7.1 SUPPORTED/UNSUPPORTED CONSTRUCTS FOR SYNTHESIS

**Q**

Which constructs are supported and unsupported by synthesis?

**A**

This question varies with the synthesizer vendor and the version. However, many synthesizers share common features. Table 7.1 summarizes the generally supported and unsupported synthesis constructs. The user must verify the synthesis rules with the tool and version of interest.

Table 7.1-1 Supported and Unsupported Synthesis Constructs

CONSTRUCT	COMMENTS AND EXAMPLE
<b>Library</b>	Supported, but generally restricted to <i>Work</i> with access to <i>IEEE</i>
<b>Packages</b>	
IEEE	<i>Std_Logic_1164, Std_Logic_Unsigned, Std_Logic_Signed, Std_Logic_Arith</i>
Standard	<i>Numeric_Bit, Numeric_Std</i> (Not yet fully adopted)      ? Supported (See types and subtypes restrictions)
TextIO	Unsupported
User package	Supported, provided the package <u>does not</u> include: -- Global signal or shared variable declarations -- Deferred constants -- Unsupported type declarations -- Unsupported constructs in subprograms (see subprograms)
<b>Entity</b>	Supported, provided port are of supported types
<b>Architecture</b>	Supported, however default binding (last compiled) takes effect.
<b>Signal, Variable, Generic</b>	Supported (Signal kind register and bus – Unsupported)
<b>Types and Subtypes</b>	<i>Integer, enumeration, Bit, Std_Logic, Bit_Vector, Std_Logic_Vector, Boolean,</i> One-dimensional array of above types Record type is either not allowed, or allowed with limitations on type of record elements and assignment of aggregates. ? Physical type – ignored or unsupported <i>Time, real, and multidimensional arrays are not allowed</i>
<b>Alias declarations</b>	Ignored or not Allowed
<b>Signal, Variable, Generic Initialization</b>	Signal and variable initialization is not allowed anywhere Constant initialized to formal parameter or to port is unsupported Generic initialization is allowed
<b>Object Classes</b>	Constants, signals and variables are allowed Files are not allowed
<b>Operators</b>	Logical (e.g. <i>and, or</i> ), relational (e.g. <i>=, /=, &gt;</i> ), concatenation, <i>[+, -]</i> (may require package), <i>[*, /, **, mod, rem]</i> (restricted to operands of power of 2 for shifts), <i>abs, not</i> , operator overloading

<b>Assignment</b>	Signal, variable, conditional signal, selected signal, and Aggregate [ e.g. (A, B, C) <= K(1 to 3) and M(1 to 3);] "AFTER clause" – IGNORED	
<b>Subprogram</b>	Supported if declared in packages or in declaration part of architect. Multiple wait statements are not allowed <i>now</i> function – Unsupported User defined resolution functions – Unsupported	
<b>Attributes</b>	Supported for <u>subset</u> of predefined attributes 'base, 'left, 'right, 'high, 'low, 'range, 'reverse_range, 'length, '[event, 'stable] allowed in <i>if</i> and <i>wait</i> statements  'pos, 'val, 'leftof, 'rightof, 'succ, 'pred, 'image, 'value, 'active, 'transaction, 'delayed, 'quiet, 'last_event, 'last_active, 'last_value Vendor's defined attributes (typically in packages) User defined attributes are unsupported	
<b>Sequential Statements</b>	- <i>Wait</i> , signal assignment (No <i>transport</i> or <i>after</i> clause), variable assignment, procedure call (with some exceptions), <i>if</i> , <i>case</i> , <i>loop</i> (restriction on loop index of integer type), <i>next</i> , <i>exit</i> , <i>return</i> , <i>null</i> - <i>Assert</i> , <i>after</i> – Ignored or unsupported	
<b>Concurrent statements</b>	Process with sensitivity list must contain all signals on the right hand side of signal assignments within the process for proper simulation. Sensitivity list is ignored.	
Process		
Concurrent signal assignment	Supported  , (guarded and transport ignored)	
Concurrent procedure	Supported  (No multiple <i>wait</i> statement allowed)	
Component Instantiation	Supported  (Type conversion on formal port not allowed)	
Generate	Supported	
BLOCK	Supported except: Block guards and register – Unsupported Ports and generics in blocks – Unsupported	
Concurrent Assertion	IGNORED or unsupported	
Configuration	Not supported	

It is important to note that the *Std\_Logic\_Arith* package is not part of the IEEE standard, and should not be assumed to be located in the IEEE library. See section 5.6 for a discussion about the availability of the IEEE *Numeric\_Std* and *Numeric\_Bit* packages. These packages define types and functions that are comparable to the *Std\_Logic\_Arith* package.

## 7.2 SYNTHESIS SENSITIVITY RULES

**Q** | In synthesis, what are the sensitivity rules for processes?

**A** | If the synthesized process has a static sensitivity list, then every read signal must be a member of this list. Otherwise the synthesis tool will create a hardware configuration that concurs with this requirement, even though the original process does not. For example, the process Original\_Lbl translates to the process Synthesis\_Lbl.

```
Original_Lbl: process(A, B)
begin
```

```
  D <= (A and B) or C;
```

```
end process Original_Lbl;
```

```
Synthesis_Lbl: process(A, B, C)
begin
```

```
  D <= (A and B) or C;
```

```
end process Synthesis_Lbl;
```

Synthesis view and re-interpretation of original process. Simulation will not agree with synthesized logic because of differences in the sensitivity list.

## 7.3 LATCH/REGISTER/COMBINATIONAL LOGIC

**Q** | What structures do arrays represent? How can they be used? What is their significance in synthesis? When are signals and variables implemented as latches, registers, or busses as outputs of combinational logic?

**A** | One dimensional arrays define vectors. In synthesis, one dimensional arrays represent either a bus, or a register, or a latch.

In clocked processes (e.g., if *Clk'event and Clk = '1'* then ... , or *wait until Clk'event and Clk = '1'*), the following rules generally apply:

1. Signals represent registers

```
Process -- clocked process
begin
  wait until Clk'event and Clk = '1';
  S <= not S;      -- S is a register
end process;
```

```
Process(Clk) -- clocked process
begin
  if Clk'event and Clk = '1' then
    S2 <= not S1; -- S2 is a register
  end if;
end process;
```

2. Reading a variable before assigning a value to that variable implies reading the "old" value of that variable. This is a register implementation for that variable.

```
Process
  variable Var : Bit;
begin
  wait until Clk'event and Clk = '1';
  Var := not Var;    -- Var is a register
  S <= Var;          -- S is a register
end process;
```

3. If a variable is assigned before it is read, and there is NO PATH where the variable is first read before it is assigned, then that variable is used as temporary storage and NO register is implied for it. If the value of that variable is later used in the equation for a signal assignment, then a register implementation is made for the assigned signal only (but not for the variable). If the variable is used in the control element (e.g., *if* or *case* statement), then the variable is used in the generation of the control logic. Therefore, to use a variable as a temporary place holder or computational element, it is important to assign a value to that variable before reading it.

```
Process
  variable Var : Bit;
begin
  wait until Clk'event and Clk = '1';
  Var := S and S2;   -- Var is combinational
  S <= Var;          -- S is a register
end process;
```

4. Signals or variables that are defined inside a clocked *if* statement and that imply registers must not be assigned a new value outside the clocked *if* statement.

```
Process (Clk, S2)
  variable Var : Bit;
begin
  if Clk'event and Clk = '1' then --clocked if statement
    Var := not Var;
    S <= Var;
  else
    Var := S3;
    S <= S2;
  end if;
end process;
```



```
Process (Clk, S3)
begin
  if Clk'event and Clk = '1' then --clocked if statement
    S <= S2;
  end if;
  S <= S3;
end process;
```



5. If a conditionally assigned variable is updated within a clocked *if* statement then that variable should not be read outside that *if* statement.

```
Process (Clk, S)
  variable Var : Bit;
begin
  if Clk'event and Clk = '1' then
    Var := not Var;
    S <= Var;
  end if;
  S2 <= Var;
end process;
```



In non-clocked processes a latch is inferred for signals or variables when one of the following conditions occurs (otherwise, the object represents a bus output of combinational logic):

1. The signal or variable is in an equivalent process where not all the alternatives of a conditional expression are considered (e.g., *if* .. *end if* construct with no *else* clause, or if all the possibilities of an *if* .. *elsif* statement have not been examined). A method to insure that a latch is not inferred in an *if* statement is to assign a value before entering the *if* statement. Another method is to avoid using the *if* statement without the *else* statement.
2. The VHDL attribute '*event*' is not present in the conditional expression.
3. The signal or variable is read in any path prior to being assigned a value (i.e., it retains the old value).

```
Process (S1)
begin
  if S1 = '1' then
    Var := not Var; -- Var is a latch
    S <= Var;        -- S is a latch
  end if;
end process;

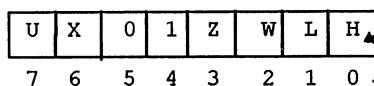
Process (S1)
begin
  Var := '0';
  S <= '0';
  if S1 = '1' then
    Var := not Var; -- Var is a combinatorial
    S <= Var;        -- S is a combinatorial
  end if;
end process;
```

Figure 7.3-1 represents a one-dimensional array model and its relation to the array index. Figures 7.3-2 and 7.3-3 represent examples of one-dimensional array declarations and their implications in synthesis.

```
variable Data_v : Std_Logic_Vector(7 downto 0) := "UX01ZWLH";
```

↑ Do not initialize in synthesis

Data\_v



Each cell can hold element of Std\_Logic

Index

Figure 7.3-1 One dimensional Array

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_arith.all;
entity Count is
    port(ResetF      : in     Std_Logic;
          Clock       : in     Std_Logic;
          Count        : out    Unsigned(3 downto 0));
end Count;
```

```
architecture Count_a of Count is
    signal Count0_s : Unsigned(3 downto 0);
    signal Count1_s : Unsigned(3 downto 0);
    signal Count2_s : Unsigned(3 downto 0);
```

```
begin
    Count0_Lbl: process(ResetF)
    begin
        if ResetF = '0' then
            Count0_s <= "0000";
        end if;
    end process Count0_Lbl;
```

*if.. end if construct with no else clause  
...hence the signal is synthesized as a latch  
because the signal must retain its value when  
the if condition is false.*

```
Count1_Lbl: process
begin
    wait until Clock'event and Clock = '1';
    if ResetF = '0' then
        Count1_s <= "0000";
    else
        Count1_s <= (Count1_s + 1);
    end if;
end process Count1_Lbl;
```

Count1\_s is in a clocked  
process, thus, the signal  
is synthesized as a  
register.

```
Count2_Lbl: process(ResetF, Count1_s)
begin
    if ResetF = '0' then
        Count2_s <= "0000";
    else
        Count2_s <= Count1_s;
    end if;
end process Count2_Lbl;
```

All if conditions are covered, and Count2\_s  
is NOT in clocked equivalent processes.  
Thus, the Count2\_s signal is synthesized as  
a combinational logic.

Count is not in a clocked process, and  
there are no conditions. Thus, it is  
synthesized as combinational logic.

```
    count <= Count1_s;
end Count_a;
```

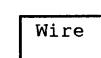


Figure 7.3-2 One-Dimensional Array Declarations, and Synthesis  
Implications (synths\count.vhd)

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_arith.all;
entity Count is
  port(ResetF      : in    Std_Logic;
        Clock       : in    Std_Logic;
        Count0      : out   Unsigned(1 downto 0);
        Count1      : out   Unsigned(1 downto 0);
        Count2      : out   Unsigned(1 downto 0));
end Count;

architecture Count_a of Count is
  signal Count1_s : Unsigned(1 downto 0);
  signal Count2_s : Unsigned(1 downto 0);
begin
  Count0_Lbl: process(ResetF)
    variable Count0_v : Unsigned(1 downto 0);
  begin
    if ResetF = '1' then
      Count0_v := Count1_s;
    end if;
    Count0 <= Count0_v;
  end process Count0_Lbl;
  Count0_v is synthesized as a latch  
because if ResetF != '0', the if  
condition is false, and Count0_v  
must retain its value.  
  Count1_Lbl: process
    variable Count1_v : Unsigned(1 downto 0);
  begin
    wait until Clock'event and Clock = '1';
    Count1_v := Count1_s; -- writing a variable before reading,  
                      -- NO register is implied.
    if ResetF = '1' then
      Count1_v := (Count1_v + 1);
    else
      Count1_v := "00";
    end if;
    Count1_s <= Count1_v; -- Register for Count1_s
  end process Count1_Lbl;
  Prior to being read, Count1_v is written with a value  
not dependent on a variable, thus the variable is a  
temporary combinational logic.  
  Count1 <= Count1_s; -- concurrent signal assignment

  Count2_Lbl: process
    variable Count2_v : Unsigned(1 downto 0);
  begin
    wait until Clock'event and Clock = '1';
    if ResetF = '1' then
      Count2_v := (Count2_v + 1); -- Reading a variable (old value),  
                                -- thus register for Count2_v
    else
      Count2_v := "00";
    end if;
    Count2_s <= Count2_v;
  end process Count2_Lbl;
  Prior to being read, Count2_v is written with a value  
dependent on a variable, thus the variable is a register
  Count2 <= Count2_s;
end Count_a;

```

Figure 7.3-3 One-Dimensional Array Declarations with Variables, and Synthesis Implications (synths\countvar.vhd)

Figure 7.3-4 represents a D Flip-flop with a synchronous reset. Figure 7.3-5 represents a D Flip-flop with an asynchronous reset.

```
entity dFF is
  port(Reset : in Bit;
        D      : in Bit;
        Clk    : in Bit;
        Q      : out Bit);
end dFF;

architecture dFF_a of dFF is
begin
  dFF_Lbl: process(Clk)
  begin
    if Clk'event and Clk = '1' then
      if Reset = '1' then
        Q <= '0';
      else
        Q <= D;
      end if;
    end if;
  end process dFF_Lbl;
end dFF_a;
```

*Reset* is in a clocked process. Thus, all inputs to this process are synchronous.

Figure 7.3-4 D Flip-flop with a Synchronous Reset (synths\dffsync.vhd)

```
architecture dFF_a of dFF is
begin
  dFF_Lbl: process(Clk, Reset)
  begin
    if Reset = '1' then
      Q <= '0';
    elsif Clk'event and Clk = '1' then
      Q <= D;
    end if;
  end process dFF_Lbl;
end dFF_a;
```

*Reset* is outside the clocked statement of the process. Thus, *Reset* is asynchronous to the clock.

Figure 7.3-5 D Flip-flop with an Asynchronous Reset (synths\dffasync.vhd)

An interesting variation to the asynchronous flip-flop is the model shown in figure 7.3-6 where two flip-flops, connected in series, have only one output flip-flop asynchronously reset. The other flip-flop is not reset. Synplify® VHDL Compiler, version 2.5e yielded the error “*The logic for q0\_s does not match a standard flip-flop.*” The code compiled with no errors when both flip-flops are reset ( $Q0\_s \leq '0'$ ). However, Synopsys® yielded the structure shown in figure 7.3-7. If  $Q0\_s$  is not reset, then it retains its old value. A multiplexer is needed, in front of the  $Q0\_s$  flip-flop input, to select either the  $D$  input (when  $Reset = '0'$ ), or the  $Q0\_s$  flip-flop output (when  $Reset = '1'$ ). This multiplexer is necessary to retain the same  $Q0\_s$  output value when a clock occurs during the assertion of the *Reset*.

```

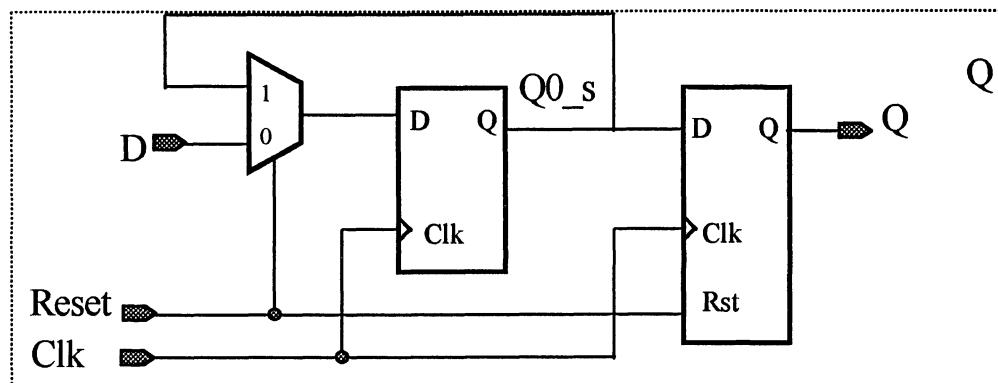
entity D2FF is
  port(Reset : in Bit;
        D      : in Bit;
        Clk    : in Bit;
        Q      : out Bit);
end D2FF;

architecture D2FF_a of D2FF is
  signal Q0_s : Bit;
begin
  D2FF_Lbl: process(Clk, Reset)
  begin
    if Reset = '1' then
      -- if Q0_s is not reset, then it retains its old
      -- value, and a multiplexer is needed in front of
      -- Q0_s Flip-Flop input to select D input when
      -- Reset = '0', or Q0_s (Flip-flop output) when
      -- Reset = '1' (to retain the same value if a clock
      -- occurs during the assertion of the Reset)
      -- Q0_s <= '0';   -- Reset of Q0_s
      Q     <= '0';
    elsif Clk'event and Clk = '1' then
      Q0_s <= D;
      Q     <= Q0_s;
    end if;
  end process D2FF_Lbl;
end D2FF_a;

```

If some, but not ALL the implied registers are not assigned a value then this code may not necessarily be synthesizable

**Figure 7.3-6 Two Sequential Flip-Flops with Asynchronous Reset of only One of Them (synths\d2ff\_asyn.vhd)**



**Figure 7.3-6 Synopsys® Implementation of Two Sequential Flip-Flops with Asynchronous Reset of only One of them**

 Avoid the partial asynchronous reset of implied registers in a clocked process. If asynchronous reset is required, either reset ALL the implied registers, or use multiple clocked processes.

*Rationale: The rule creates more portable models. Unexpected additional multiplexers may be implemented if the rule is violated.*

## 7.4 LATCH INFERRANCE IN FUNCTIONS

**Q** | Can latches be inferred by functions?

**A** |  Since variables declared in a function do not retain their values between calls to this function, latches will not be inferred by synthesis (even if the aforementioned latching criteria are satisfied). To avoid any confusion, the function should be written in a non-latching coding style. Consequently, the inferred hardware will agree with the basic VHDL synthesis coding styles.

## 7.5 VARIABLE INITIALIZATION AND LIFETIME

**Q** | When are variables initialized? What is the lifetime of variables? Do they persist throughout the simulation, or do they become re-initialized to their default values on each execution of the process or subprogram?

**A** | A variable is initialized when, and only when, its declaration is elaborated. Variable initialization in processes and subprograms is discussed in the following subsections.

### 7.5.1 Variable Initialization in Processes

A declaration in a process statement is elaborated once when the process is elaborated. This occurs during the static elaboration phase of simulation, before the execution phase. A variable in a process statement retains its value while a process is suspended. Section 7.3 discusses how, in synthesis, variables are implemented as combinational logic, or as storage elements.

A concurrent procedure is equivalent to a process, with a procedure call followed by an explicit *wait on* sensitivity list. This sensitivity list is extracted from all the actual signals whose mode in the formal parameter list is *in* or *inout*. Initialization of variables declared in concurrent procedures is equivalent to the initialization of variables declared in processes.

 In synthesis, **DO NOT** initialize variables in the declaration statements of the variables. This rule applies regardless of where the variables are declared, in processes or subprograms. Instead, declare the variable UNINITIALIZED, and in the body of the process or subprogram, write an initialization statement (e.g., *My\_Variable := 0;*). Also in synthesis, **DO NOT** declare constants that are initialized to values of formal parameters or ports. See figure 7.5.1-1 and 7.5.1-2 for examples.

*Rationale:* Initializing a variable would lead to a mismatch between simulation and synthesis.

```

entity InitVarCnst is
  port (A : in Bit;
        B : out Bit;
        C : out Bit);

end InitVarCnst;

architecture InitVarCnst_NonSynth of InitVarCnst is
  constant K : Bit := A; -- Initialization not allowed 
  function Invert (I : Bit) return Bit is
    constant W : Bit := I; -- Initialization not allowed 
  begin
    return not W;
  end Invert ;

begin -- InitVarCnst_NonSynth
  B <= Invert(K);

  P_Lbl : process (A)
    constant P : Bit := A; -- Initialization not allowed 
    begin -- process P_Lbl
      C <= not P;
    end process P_Lbl;
  end InitVarCnst_NonSynth;

```

Figure 7.5.1-1 Non-Synthesizable VHDL Code (synths\initvc1.vhd)

```

architecture InitVarCnst_OK of InitVarCnst is
  signal K : Bit; -- signal declared, but not initialized 
  function Invert (I : Bit) return Bit is
    variable W : Bit; -- Variable declared, but not initialized 
  begin
    W := I; -- Variable initialized 
    return not W;
  end Invert;
begin -- InitVarCnst_OK
  K <= A;

  B <= Invert(K);

  P_Lbl : process (A)
    variable P : Bit; -- Variable declared, but not initialized 
    begin -- process P_Lbl
      P := A;
      C <= not P;
    end process P_Lbl;
  end InitVarCnst_OK;

```

Figure 7.5.1-1 -Synthesizable VHDL Code (synths\initvc2.vhd)

### 7.5.2 Variable Initialization for Subprograms (Called from Processes)

A declaration in a subprogram body is elaborated each time the subprogram is called. For procedures, the variables retain their value until the procedure returns. If the procedure executes a *wait* statement, the calling process is suspended and the procedure variables retain their values. When the process resumes, the procedure continues executing with the retained variable values. If the procedure returns, the local variables declared in the procedure are no longer in existence. If the procedure is called again (regardless of whether a calling process suspends or not), the variables are effectively re-initialized.

A process with a sensitivity list cannot contain any explicit *wait* statement, and any called procedure from such a process cannot contain a *wait* statement.

Functions cannot contain *wait* statements. Therefore, variable declared in functions are initialized each time the functions are called.

As mentioned in the previous subsection, when a model is targeted for synthesis, variables should not be initialized in subprograms.

## 7.6 WAIT STATEMENT

**Q** | Where can the *wait* statement be used in synthesis, and what are its restrictions?

**A** | The *wait* statement can be used anywhere in a process except in a *for...loop* statement or in a subprogram. In most synthesizers, there can only be a single *wait* statement in a process. A *wait* statement in a *for...loop* implies multiple *wait* statements, one for each iteration of the loop. A subprogram can be called multiple times in process. Therefore, allowing a *wait* statement in a subprogram would imply multiple *wait* statements, if that subprogram is called multiple times from within a process. If a subprogram is called as a concurrent procedure, then there is an implied *wait* statement at the end of that procedure (sensitive to all formal signals of mode *in* or *inout*). Adding a *wait* statement in a concurrent procedure would then include two *wait* statements – the explicit *wait* and the implicit *wait*. Multiple *wait* statements are only allowed in behavioral synthesis (see section 7.27).

*If any logic path has a wait statement, then all logic paths (following the wait) must also have a wait statement. Consider the example shown in figure 7.6-1. When this code is synthesized, all the logic paths have a wait statement, and are synchronous to the clock.* Figure 7.6-2 represents the results of synthesis using QuickWorks Toolkit for QuickLogic<sup>12</sup> for FPGAs using Synplify-Lite®<sup>13</sup> synthesizer.

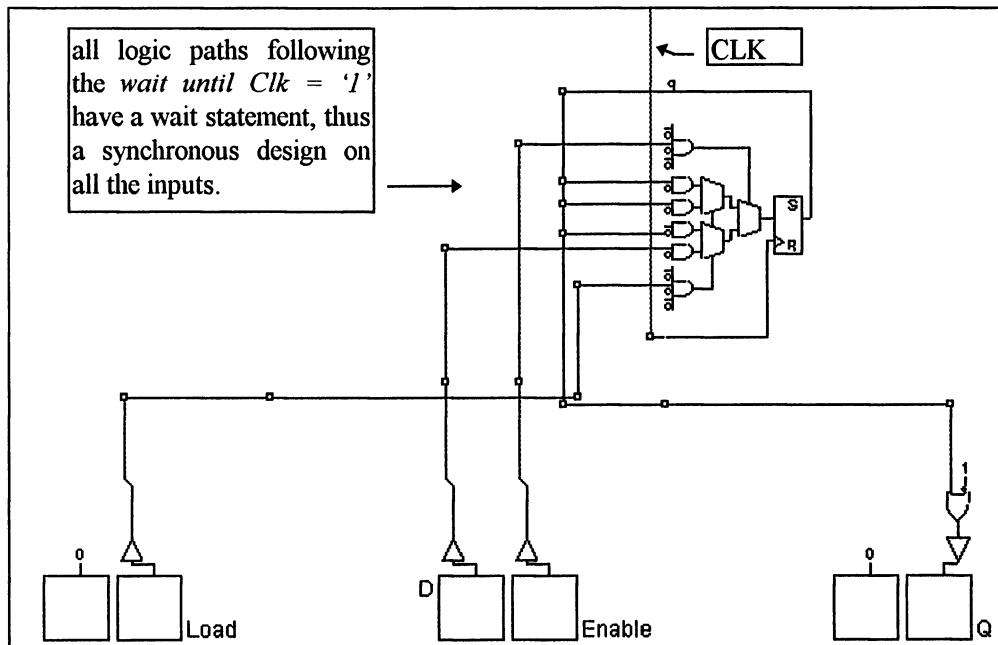
```

entity dFF is
  port(Enable : in Bit;
        Load   : in Bit;
        D      : in Bit;
        Clk    : in Bit;
        Q      : out Bit);
end dFF;

architecture dFF_a of dFF is
  signal Q_s : Bit;
begin
  dFF_Lbl: process
  begin
    wait until Clk = '1';
    if Enable = '1' and Load = '0' then
      Q_s <= not Q_s;
    elsif Enable = '1' and Load = '1' then
      Q_s <= D;
    end if;
  end process dFF_Lbl;
  Q <= Q_s;
end dFF_a;

```

**Figure 7.6-1 Flip-Flop Model (synths\dff.vhd)**



**Figure 7.6-1 Synthesized Flip-Flop for FPGA Implementation**

## 7.7 DEFINING SHIFT REGISTERS IN SYNTHESIS

**Q**

Given the code shown in figure 7.7 for a shift register, Synopsys® yields an elaborate Error "Tried to use a synchronized value." Where is the error?

```
entity Register9 is
    port (Reset : in Bit;
          Clk : in Bit;
          Data : in Bit;
          RegOut : out Bit_Vector(8 downto 0));
end Register9;

architecture Register_a of Register9 is
    signal Temp : Bit_Vector(8 downto 0);
begin
    process(Reset, Clk, Data)
        variable Reg : Bit_Vector(8 downto 0);
    begin
        if (Reset = '0') then
            for I in 8 downto 0 loop
                Reg(I) := '0';
            end loop;
        elsif (Clk = '1' and Clk'event) then
            for I in 8 downto 1 loop
                Reg(I) := Reg(I - 1);
            end loop;
            Reg(0) := Data;
        end if;
        Temp <= Reg; -- ===== Error line
    end process;
```



Figure 7.7 Synthesis Error (Synths\reg9.vhd)

**A**

The code has several errors that will confuse the synthesizer. Before attempting to point to the errors, it is important to understand the concepts of register and latch

inferences. Registers are always inferred when signals are assigned values in a clocked process (e.g., a process with a *wait until Clk'event and Clk = '1'*; statement). The rules for register implementation of variables are different as discussed in the previous section.

### 7.7.1 Resolution -- Code for Shift Register

The code in the original question contains several errors as identified in figure 7.7.1-1.

1. *Data* should NOT be in the sensitivity list; it is not used to control the operation of the process (like the *Reset* or the *Clk*). Specifically, no decision is made based on the value of the data (no *if data = '0' then* statement).
2. Variable *Reg* implies a register; the variable reads an old value retained from a previous clock (*Reg(I) := Reg(I-1);*).
3. *Temp* is a signal, and its value is retained between clocks. It implies a register. This model represents two registers, register *Temp* being driven by register *Reg*. This is not the original intent of the code.

```

----.
signal Temp : Bit_Vector(8 downto 0);

process(Reset, Clk, Data) --  

    variable reg : Bit_Vector(8 downto 0);
begin
    if (Reset = '0') then
        for I in 8 downto 0 loop
            Reg(I) := '0';
        end loop;
    elsif (Clk = '1' and Clk'event) then
        for I in 8 downto 1 loop
            Reg(I) := Reg(I-1); --  

        end loop;
        Reg(0) := Data;
    end if;
    Temp <= Reg; --  

end process;

```

"Data" should NOT be in the sensitivity list

Reg(I) reads Reg(I-1) before Reg is assigned a value, thus Reg is a register. Temp is a signal, implemented as a register. Thus 2 registers!!!

Figure 7.7.1-1 Code Errors for Synthesis

Figure 7.7.1-2 represents a modification to this code, while maintaining the same structure. The variable *Reg*, in the original architecture, is changed to a signal in this modified architecture.

```

Entity RegisterOK is
    port (Reset : in Bit;
          Clk : in Bit;
          Data : in Bit;
          RegOut : out Bit_Vector(8 downto 0));
end RegisterOK;

architecture RegisterOK_a of RegisterOK is
    signal Reg : Bit_Vector(8 downto 0);
begin
    process(Reset, Clk) --Data should NOT be in the sensitivity list
    begin
        if (Reset = '0') then
            for I in 8 downto 0 loop
                Reg(I) <= '0';
            end loop;
        elsif (Clk = '1' and Clk'event) then
            for I in 8 downto 1 loop
                Reg(I) <= Reg(I-1);
            end loop;
            Reg(0) <= Data;
        end if;
    end process;

    RegOut <= Reg;
end RegisterOK_a;

```

Figure 7.7.1-2 Corrected Shift Register model, (synth\regok.vhd)

Figure 7.7.1-3 represents an improved shift register model. It does not use loops since concatenation can achieve the same effect in this case. This model is more generic and reusable as it uses attributes to identify the size of *Reg*, without the necessity to change the code.

```

Entity RegImproved is
    generic (Width_g : Natural := 9); ← Generic enhanced
    port (Reset : in Bit; flexibility and
          Clk : in Bit;
          Data : in Bit;
          RegOut : out Bit_Vector(Width_g -1 downto 0));
end RegImproved;

architecture RegImproved_a of RegImproved is
    signal Reg : Bit_Vector(Width_g -1 downto 0);
begin
    Reg_Lbl: process(Reset, Clk)
        begin
            if (Reset = '0') then
                Reg <= (others => '0');
            elsif (Clk = '1' and Clk'event) then
                Reg <= Reg(Reg'left-1 downto Reg'right) & Data;
            end if;
        end process Reg_Lbl;
        RegOut <= Reg;
    end RegImproved_a;

```

**Figure 7.7.1-3 Improved Shift Register (synths\regok2.vhd)**

## 7.8 REGISTER FILE

**Q** | How can a register file, *n* words deep by *m* bits wide be defined?

### 7.8.1 Register File – with signals

**A** | Generally, multi-dimensional arrays are not allowed in synthesis for signal or variable objects. They are allowed as constants or table lookups. A solution is to declare two one-dimensional array types. This approach is easier to use and more representative of actual hardware. Figure 7.8.1 demonstrates the declaration of a register file. Many synthesis tool vendors and chip vendors provide libraries of available high level parameterized components targeted toward their specific technologies. The list includes components like RAM, Counters, adders, multipliers, FIFOs, Shift registers, linear feedback shift registers, pipeline registers multiplexers, decoders, parity generators, and megafunctions (e.g., DMA controller, interrupt controller, microprocessor, etc.). If these components are available, they should be used instead of the VHDL user defined model.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_arith.all;
entity Register_Nty is
  generic (DataWidth_g : Natural := 8;
           AddrWidth_g : Natural := 2);
  port(
    RegAddr   : in  Std_Logic_Vector(AddrWidth_g - 1 downto 0);
    InData    : in  Std_Logic_Vector(DataWidth_g - 1 downto 0);
    OutData   : out Std_Logic_Vector(DataWidth_g - 1 downto 0);
    Clk       : in  Std_Logic;
    RdWrF    : in  Std_Logic);
end Register_Nty;
architecture Register_a of Register_Nty is
  subtype Data_Typ is Std_Logic_Vector(DataWidth_g - 1 downto 0);
  subtype RegSize_Typ is integer range 0 to 2 ** AddrWidth_g - 1;
  type Reg_Typ is array(RegSize_Typ) of Data_Typ;
  signal Register_s : Reg_Typ;
begin -- Register_a
  Write_Lbl : process
  begin -- process Register_Lbl
    wait until Clk'event and Clk = '1';
    if RdWrF = '0' then
      Register_s(CONV_INTEGER(Unsigned(RegAddr))) <= InData;
    end if;
  end process Write_Lbl ;
  Read_Lbl: process (RdWrF, RegAddr)
  begin
    if RdWrF = '1' then
      OutData <= Register_s(CONV_INTEGER(Unsigned(RegAddr)));
    else
      OutData <= (others => 'Z');
    end if;
  end process Read_Lbl;
end Register_a;

```

**Signal Register\_s is an implied register because it is updated in a clocked process.**

**Port Outdata is an output of combinatorial logic (tri-stated in this case) because this signal is updated under ALL conditions, and it is NOT in a clocked process.**

Figure 7.8.1 Register File model (synths\regfile.vhd)

### 7.8.2 Register File – with variables

The model described in figure 7.8.1 makes use of a signal for the declaration of the register file. However, the simulator memory overhead associated with signals is very taxing. Specifically, each signal typically costs 100 bytes of additional overhead (over variables). Thus, a 16 bit wide by 100 deep register file requires 160 MBytes of additional simulation memory overhead ( $16 * 100$  (for register file depth) \* 100 (for overhead)). This overhead can be minimized by declaring the register file as a variable rather than a signal. However, as explained in section 7.2, care must be taken to prevent the unwanted creation of additional latches of registers. Figure 7.8.2-1 demonstrates an improper architecture of a register file using a variable declaration. In that architecture, *OutData* is implemented as a register because it is in a clocked process and *OutData* is NOT updated under ALL the conditions of the *if* statement. This is not the intended design. Figure 7.8.2-2 demonstrates the correct use of such an architecture where *OutData* is updated under all the conditions of the *if* statement.

```

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.std_logic_arith.all;

entity RegFile is
generic (DataWidth_g : Natural := 8;
          AddrWidth_g : Natural := 2);
port(
  RegAddr      : in  Std_Lock_Vector(AddrWidth_g - 1 downto 0);
  InData       : in  Std_Lock_Vector(DataWidth_g - 1 downto 0);
  Clk          : in  Std_Lock;
  RdWrF        : in  Std_Lock;
  OutData      : out Std_Lock_Vector(DataWidth_g - 1 downto 0));
end RegFile;

architecture RegFile_a of RegFile is
subtype Data_Typ is Std_Lock_Vector(DataWidth_g - 1 downto 0);
subtype RegSize_Typ is Integer range 0 to 2 ** AddrWidth_g - 1;
type Reg_Typ is array(RegSize_Typ) of Data_Typ;
begin -- RegFile_a
  Register_v : process
    variable Register_v : Reg_Typ;
  begin -- process Register_v
    wait until Clk'event and Clk = '1';
    if RdWrF = '0' then
      Register_v(CONV_INTEGER(Unsigned(RegAddr))) := InData;
    else --- RdWrF = '1'
      OutData <= Register_v(CONV_INTEGER(Unsigned(RegAddr)));
    end if;
  end process RdWrite_Lbl;
end RegFile_a;

```

Register file declared as a variable

Variable *Register\_v* is synthesized as registers because it is in a clocked process and it is NOT updated under ALL the conditions of the *if* statement.

\* Output *OutData* is synthesized as registers because it is in a clocked process and it is NOT updated under ALL the conditions of the *if* statement.

Figure 7.8.2-1 Improper Register File with Variable (synths\regf\_bad.vhd)

```

architecture RegFile_a of RegFile is
  subtype Data_Typ is Std_Logic_Vector(DataWidth_g - 1 downto 0);
  subtype RegSize_Typ is Integer range 0 to 2 ** AddrWidth_g - 1;
  type Reg_Typ is array(RegSize_Typ) of Data_Typ;

begin -- RegFile_a

  RdWrite_Lbl : process(Clk, RdWrF, RegAddr)
    variable Register_v : Reg_Typ;

  begin -- process Register_Lbl
    if RdWrF = '1' then -- READ
      OutData <= Register_v(CONV_INTEGER(Unsigned(RegAddr)));
    else
      OutData <= (others => 'Z');
    end if;
    -- If OutData is always read, then
    -- do not use the "if" statement.
    -- Use the following instead:
    --
    -- OutData <= Register_v(CONV_INTEGER(Unsigned(RegAddr)));

    if Clk'event and Clk = '1' then
      if RdWrF = '0' then
        Register_v(CONV_INTEGER(Unsigned(RegAddr))) := InData;
      end if;
    end if;
  end process RdWrite_Lbl;
end RegFile_a;

```

Output *OutData* is synthesized as combinatorial logic because it is updated under ALL the conditions of the *if* statement.

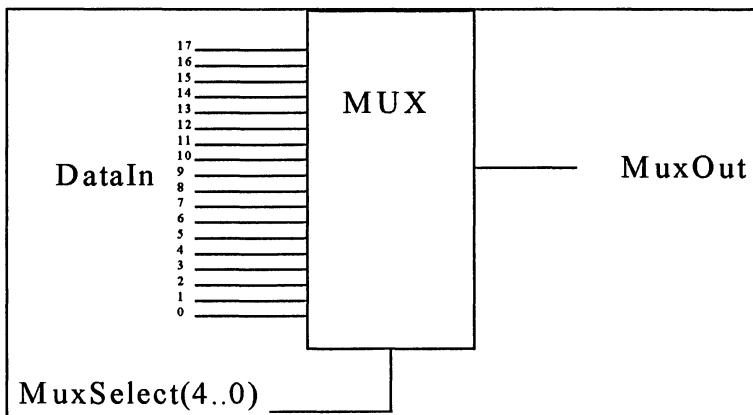
Variable *Register\_v* is synthesized as registers because it is in a clocked process and it is NOT updated under ALL the conditions of the *if* statement.

**Figure 7.8-3 Register File with Variable (synth\regfilev.vhd)**

## 7.9 MULTIPLEXER MODEL

**Q** Q: How can an 18 bit multiplexer be specified, using a concurrent signal assignment, without the use of processes?

**A** A | Figure 7.9 demonstrates the code for such a multiplexer.



```

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.Std_Lock_Arith.all;

entity Mux is
    port(DataIn      : in  Std_Lock_Vector(17 downto 0);
          MuxSelect   : in  Std_Lock_Vector( 4 downto 0);
          MuxOut      : out Std_Lock);
end Mux;

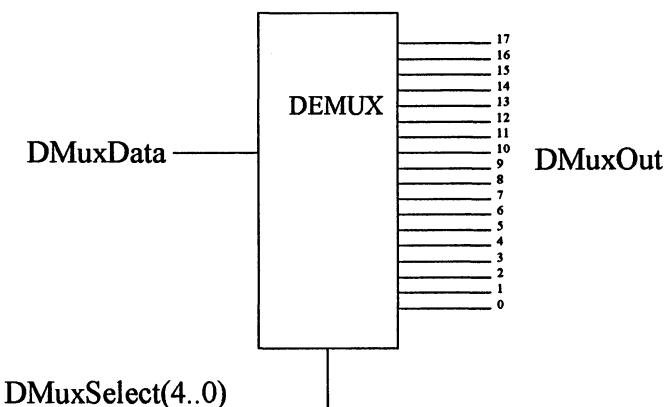
architecture Mux_a of Mux is
begin
    -- Multiplexer 18 to 1
    MuxOut <= DataIn(Conv_Integer(Unsigned(MuxSelect))) when
               MuxSelect < "10010"
               'X';
end Mux_a;

```

**Figure 7.9 Multiplexer Model (synths\mux.vhd)**

## 7.10 DEMULTIPLEXER MODEL

- Q** How can an 18 bit demultiplexer be specified using a concurrent signal assignment, without the use of processes? The demultiplexer accepts a single bit, and passes that value onto one of eighteen outputs under the control of five select lines. All unselected output should have a value of '0'.
- A** Several approaches can be used. Figure 7.10-1 demonstrates the use of concurrent statements. Another, more elegant method, uses a loop in a process as shown in figure 7.10-2. A third, and preferred approach makes a signal assignment (to '0') on all the bits of the output, but updates the required output. The use of a loop is bypassed (see figure 7.10-3).



```

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.Std_Lock_Arith.all;
entity Demux is
    port(DMuxSelect : in  Std_Lock_Vector(4 downto 0);
          DMuxData   : in  Std_Lock;
          DMuxOut    : out Std_Lock_Vector(17 downto 0));
end Demux;

architecture Demux_a of Demux is
    signal DmuxCntrl : Integer range 0 to 17;
begin
    -- Demux control for computation as an integer index
    DmuxCntrl <= 0 when Conv_Integer(Unsigned(DmuxSelect)) > 17 else
        Conv_Integer(Unsigned(DmuxSelect));
    DMuxOut(0)  <= DmuxData when DmuxCntrl = 0 else '0';
    DMuxOut(1)  <= DmuxData when DmuxCntrl = 1 else '0';
    DMuxOut(2)  <= DmuxData when DmuxCntrl = 2 else '0';
    DMuxOut(3)  <= DmuxData when DmuxCntrl = 3 else '0';
    DMuxOut(4)  <= DmuxData when DmuxCntrl = 4 else '0';
    DMuxOut(5)  <= DmuxData when DmuxCntrl = 5 else '0';
    DMuxOut(6)  <= DmuxData when DmuxCntrl = 6 else '0';
    DMuxOut(7)  <= DmuxData when DmuxCntrl = 7 else '0';
    DMuxOut(8)  <= DmuxData when DmuxCntrl = 8 else '0';
    DMuxOut(9)  <= DmuxData when DmuxCntrl = 9 else '0';
    DMuxOut(10) <= DmuxData when DmuxCntrl = 10 else '0';
    DMuxOut(11) <= DmuxData when DmuxCntrl = 11 else '0';
    DMuxOut(12) <= DmuxData when DmuxCntrl = 12 else '0';
    DMuxOut(13) <= DmuxData when DmuxCntrl = 13 else '0';
    DMuxOut(14) <= DmuxData when DmuxCntrl = 14 else '0';
    DMuxOut(15) <= DmuxData when DmuxCntrl = 15 else '0';
    DMuxOut(16) <= DmuxData when DmuxCntrl = 16 else '0';
    DMuxOut(17) <= DmuxData when DmuxCntrl = 17 else '0';
end Demux_a;

```

**Figure 7.10-1 Demultiplexer Model with Concurrent Statements  
(synths\demuxc.vhd)**

```

architecture DemuxBetter_a of Demux is
    signal DmuxCntrl : Integer range 0 to 17;
begin
    -- Demux control for computation as an integer index
    DmuxCntrl <= 0 when Conv_Integer(Unsigned(DmuxSelect)) > 17 else
        Conv_Integer(Unsigned(DmuxSelect));

    T_Lbl: process (DmuxCntrl, DMuxData)
    begin
        for I in DMuxOut'range loop
            if DmuxCntrl = I then
                DMuxOut(I)  <= DmuxData;
            else
                DMuxOut(I)  <= '0';
            end if;
        end loop;
    end process T_Lbl;
end DemuxBetter_a;

```

**Figure 7.10.2 Demultiplexer Model with a Process (synths \demuxp1.vhd)**

```

architecture DemuxV2_a of Demux is
    signal DmuxCntrl : Integer range 0 to 17;
begin
    -- Demux control for computation as an integer index
    DmuxCntrl <= 0 when Conv_Integer(Unsigned(DmuxSelect)) > 17 else
        Conv_Integer(Unsigned(DmuxSelect));

    Dmux_Lbl: process (DmuxCntrl, DMuxData)
    begin
        DMuxOut <= (others => '0');
    end process Dmux_Lbl;
end DemuxV2_a;

```

All bits are reset to '0'

Selected bit is set. Last signal assignment takes effect.

**Figure 7.10-3 Demultiplexer Model with a Process, no Loop  
(synths\demuxp2.vhd)**

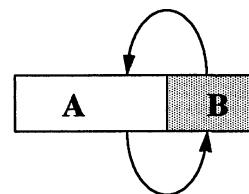
## 7.11 BARREL SHIFTER

**Q**

Why does this legal VHDL model of a barrel shifter (shown in Figure 7.11-1) fail to synthesize? The following error is flagged:

Error: Constant value required in routine Barrel\_Shift line 15

In a barrel shift, bits B are exchanged with bits A. The amount of shift determines the ranges for these bits. A and B make up the inputs to the barrel shifter.



```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity Barrel_Shift is
    generic(Width_g : Natural := 32);
    port(D_in : in Std_Logic_Vector(Width_g - 1 downto 0);
          D_out : out Std_Logic_Vector(Width_g - 1 downto 0);
          Amount : in Integer range 0 to Width_g - 1);
end Barrel_Shift;

architecture Barrel_Shift_a of Barrel_Shift is
begin
    Barrel_Lbl: process(D_in, Amount)
    begin
        D_out <= D_in((D_in'left - Amount) downto 0) & ← Error Line ⚡*
            D_in(D_in'left downto (D_in'left - Amount + 1));
    end process Barrel_Lbl;
end Barrel_Shift_a;

```

Region B

Region A

**Figure 7.11-1 Non-Synthesizable Barrel Shifter (synths\barrelnc.vhd)**

**A**

This model has two synthesis problems:

1. *The slice “D\_in((D\_in'left - Amount) downto 0) & D\_in(D\_in'left downto (D\_in'left - Amount + 1));” is non-computable by the synthesizer since “Amount” is a dynamic value that can change in value. In Synopsys, only computable slices may be synthesized.*
2. The vector “D\_in(D\_in'left downto (D\_in'left - Amount + 1));” is a null array (in VHDL’87) when *amount* is zero.

Figure 7.11-2 provides a synthesizable model. Notice that the loop counter is compared to the input *Amount*. If they are equal, then the barrel shift occurs as determined by the amount of the loop counter (a computable value). During synthesis, the loop is unraveled or expanded flat, and then the appropriate equations are produced and minimized.

```
architecture Barrel_Shift_Synth of Barrel_Shift is
begin
  Barrel_Lbl: process(D_in, Amount)
    variable DIn_v : Std_Logic_Vector(D_In'length - 1 downto 0);
  begin
    DIn_v := D_In;
    if Amount = 0 then
      D_Out <= DIn_v;           -- Slice is computable because of comparison to the loop index.
                                -- Null slice is also avoided because test for Amount = 0 is first
                                -- performed and handled outside the loop.
    else
      for Lp in DIn_v'length - 1 downto 1 loop
        if Lp = Amount then -- found amount of shift
          D_out <= DIn_v((DIn_v'left - Lp) downto 0) &
                        DIn_v(DIn_v'left downto (DIn_v'left + 1 - Lp));
        end if;
      end loop;
    end if;
  end process Barrel_Lbl;
end Barrel_Shift_Synth;
```

**figure 7.11-2 Synthesizable Model of a Barrel Shifter (synths\barlsyn1.vhd)**

Figure 7.11-3 demonstrates another approach in the modeling of a barrel shifter. In this example, a *case* statement is used with all cases of the shift amount explored.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity Barrel_Shift is
  port(D_in   : in  Std_Logic_Vector(7 downto 0);
       D_out  : out Std_Logic_Vector(7 downto 0);
       Amount : in  Integer range 0 to 7);
end Barrel_Shift;
```

```

architecture Barrel_Shift_Synth of Barrel_Shift is
begin
    Barrel_Lbl: process(D_in, Amount)
    begin
        case Amount is
            when 0 => D_Out <= D_In;
            when 1 => D_Out <= D_In(6 downto 0) & D_In(7);
            when 2 => D_Out <= D_In(5 downto 0) & D_In(7 downto 6);
            when 3 => D_Out <= D_In(4 downto 0) & D_In(7 downto 5);
            when 4 => D_Out <= D_In(3 downto 0) & D_In(7 downto 4);
            when 5 => D_Out <= D_In(2 downto 0) & D_In(7 downto 3);
            when 6 => D_Out <= D_In(1 downto 0) & D_In(7 downto 2);
            when 7 => D_Out <= D_In(0) & D_In(7 downto 1);
        end case;
    end process Barrel_Lbl;
end Barrel_Shift_Synth;

```

**Figure 7.11-3 Synthesizable Barrel Shifter with Case Statement,  
(synths\barrel.vhd)**

## 7.12 USE OF "DON'T CARE" IN CASE STATEMENT

**Q** Using the following function that checks the leading one position, how can "don't care" be expressed? The following code does not work. The use of the *if* seems inefficient for synthesis.

```

function Leading_One(S : Std_Logic_Vector(3 downto 0))
                    return Integer is
begin
    case S is
        when "1---" => return 3;
        when "01--" => return 2;
        when "001-" => return 1;
        when others => return 0;
    end case;
end;

```

**A** There is a difference between a simulation state ("don't care" state) versus "don't care", or ignore meaning. In VHDL there are nine discrete states for a signal of type *Std\_Logic*:

'U'	-- Uninitialized	'Z'	-- High Impedance
'X'	-- Forcing Unknown	'W'	-- Weak Unknown
'0'	-- Forcing 0	'L'	-- Weak 0
'1'	-- Forcing 1	'H'	-- Weak 1
		'-'	-- Don't care

In VHDL "don't care" means '-' state, not 'U' or '1' or anything else. This maybe useful during simulation to ensure that a signal is not resolved with any other signal. The *IEEE.Std\_Logic\_1164* resolution function resolves a '-' with anything but a 'U' to an 'X'. It also resolves a 'U' with anything (including '-') to a 'U'.

The code mentioned in the problem does not work because '-' compared to '0' (or compared to '1') yields FALSE. In synthesis, the "don't care" value of *Std\_Logic* should be used in

assignments only (i.e., S <= "1--"), not in expressions to be used for comparisons. VHDL tools check equivalence by comparing strings; therefore, the only string that will match "1--" is "1--". This case statement compares just the *Std\_Logic\_Vector* S against the applied constants. Only if S = "1---", then the return value will be 3. It will not be 3 for "10--" or "1x--" or any other value.

The following subsections define potential approaches for expressing the “don’t care” intention. Also see section 7.13 for a description of a priority encoder model.

### 7.12.1 Case Statement with No “Don’t Care”

The cases can be individually expressed so that all the pertinent cases are covered. This is demonstrated in figure 7.12.1-1.

```

library IEEE;
use IEEE.std_logic_1164.all;

entity Leading_One_Test is
  port( S : in Std_Logic_Vector(3 downto 0);
        T : out Integer range 0 to 3 );
end;

architecture LeadOne_a of Leading_One_Test is
  function Leading_One(S : Std_Logic_Vector(3 downto 0))
    return Integer is
begin
  case S is
    when "1000" | "1001" | "1010" | "1011" |
      "1100" | "1101" | "1110" | "1111" => return 3;
    when "0100" | "0101" | "0110" | "0111" => return 2;
    when "0010" | "0011" => return 1;
    when others => return 0;
  end case;
end;
begin
  T <= leading_one(S);
end LeadOne_a;

```

Range limits synthesis to produce a 2-bit result instead of a 32-bit result.

"others" branch is needed not only to cover the cases "0001" or "0000", but also all cases including any meta value like '-' or 'X'.

**Figure 7.12.1-1 Using Case Statements to Cover All Conditions,  
(synths\lead1.vhd)**

Figure 7.12.1-2 demonstrates another use of a nested *case* statement. This may not be as elegant as a few lines of code, but it preserves the *case* statement philosophy and gets the job done. In one synthesizer, it synthesized into five gates. (Two inverters and three *nand* gates).

```

architecture LeadOneTest_a of LeadOneTest is
begin
  process (S)
  begin
    case S(3) is
      when '1' => T <= 3;
      when '0' =>
        case S(2) is
          when '1' => T <= 2;
          when '0' =>
            case S(1) is
              when '1' => T <= 1;
              when others => T <= 0;
            end case;
            when others => T <= 0;
          end case;
          when others => T <= 0;
        end case;
      end case;
    end process;
  end LeadOneTest_a;

```

**Figure 7.12.1.-2 Nested Case Statement (synths\lead2.vhd)**

Figure 7.12.1.-3 represents yet another representation of the *case* statement that uses type *Unsigned* defined in package *Numeric\_Std*. This approach converts the input into an *integer*, and performs the case alternatives on ranges of integer values. In synthesis, the *case* statement causes an equalized priority in the selection of the case select object.

```

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.Numeric_Std.all;

entity test is
  port (S : in Unsigned(3 downto 0);
        T : out Integer range 0 to 3);
end test;
architecture Testcase of Test is
begin
  process(S)
  begin
    case To_Integer(S) is
      when 8 to 15 => T <= 3;
      when 4 to 7 => T <= 2;
      when 2 to 3 => T <= 1;
      when others => T <= 0;
    end case;
  end process;
end Testcase;

```

In synthesis, *case* statement causes the same priority in the selection of the *S* signal. Constraints can be used to optimize the priorities of signals.

Use *CONV\_INTEGER* function if *IEEE.Std\_Lock\_Arith* is used instead of *IEEE.Numeric\_Std*.

Good approach in use of integers with range

**Figure 7.12.1.-3 Case Statement with Type "Unsigned" and Package Numeric\_Std (synths\lead3.vhd)**

### 7.12.2 IF Statement

Shorter code can be obtained by using the *if...elsif* construct. *If* statements are not necessarily synthesized less efficiently than *case* statements. Figure 7.12.2 demonstrates the use of the *if* statement. Signals that make up the conditional expression, and that are near the top of the *if* construct, are considered to have higher priority than later expressions. Synopsys® will place those signals closer to the output than signals that are used later in the expression. Note that synthesizers handle the *if* and *case* statements differently because the conditionals are mutually exclusive (by definition) for the *case* statement, whereas they are not mutually exclusive in the *if* statement.

```
architecture Testif of Test is
begin
  process(S)
  begin
    if S(3)='1'
      then T <= 3;
    elsif S(2)='1'
      then T <= 2;
    elsif S(1)='1'
      then T <= 1;
    else T <= 0;
    end if;
  end process;
end testif;
```

S(3) given higher  
priority than S(2).  
Thus, S(3) can have a  
slower arrival time.

Figure 7.12.2 Using *IF* instead of *Case* Statements (synths\lead4.vhd)

### 7.12.3 Loop Statement

Another approach in the implementation of a function to check the leading one position is the use of a loop. The algorithm shown in figure 7.12.3-1 and 7.12.3-2 uses a *for...loop* to determine when the index of the first bit of *S* is '1' (from the left to the right bound of the index range). This function synthesizes to purely combinatorial logic with Synopsys® if *S'range* is static (i.e., known at compile time). This approach is general because the loop works with any range, and is not limited to a fixed range.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity leading_one_test is
  generic(Width_g : Natural := 4);
  port( S : in Std_Logic_Vector(Width_g - 1 downto 0);
        T : out Integer range 0 to Width_g - 1);
end;
```

```

architecture Leading_One_Test_a of Leading_One_Test is
  function Leading_One( s : Std_Logic_Vector ) return Integer is
    variable Leading_One_Position : Integer;
    variable S_v : Std_Logic_Vector(S'length - 1 downto 0);
  begin
    S_v := S;
    LP_Lbl: for I in S_v'range loop
      Leading_One_Position := I;
      if S_v(I) = '1' then
        exit LP_Lbl;
      end if;
    end loop;
    return Leading_One_Position;
  end;

begin
  T <= Leading_One(S);
end Leading_One_Test_a;

```

**It's very IMPORTANT to normalize the range of the loop to control the range and direction of the subprogram local variables.**

**Rationale:** The subprogram may lead to errors if the range of the actual parameter is ascending and no normalization is performed.

Figure 7.12.3-1 Using “for.. loop” in a Function (synths\lead5.vhd)

```

architecture Leading_One_Test_b of Leading_One_Test is
begin
  process(S)
    variable S_v : Std_Logic_Vector(Width_g - 1 downto 0);
  begin
    S_v := S;
    LP_Lbl: for Lp in S_v'range loop
      T <= Lp;
      exit LP_Lbl when S_v(Lp) = '1';
    end loop;
  end process;
end Leading_One_Test_b;

```

Not all vendors support the `exit` statement (see 7.13.1).

Figure 7.12.3-2 Using *for.. loop* in a Process (synths\lead6.vhd)

#### 7.12.4 IEEE Std\_Synth

The *Numeric\_Std* package (see section 5.6 for availability) provides the overloaded function *STD\_MATCH* as shown below. This function interprets the '-' character into a true “don’t-care”. However, not many synthesis vendors have yet adopted this template, as defined in the package. Figure 7.12.4 demonstrates the application of this function.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;

entity test is
  port (I : in Unsigned(3 downto 0);
        T : out Integer range 0 to 3);
end test;

```

```

architecture TestMatch of Test is
begin
  process(I)
  begin
    if STD_MATCH(I, "1--") then
      T <= 3;
    elsif STD_MATCH(I, "-1-") then
      T <= 2;
    elsif STD_MATCH(I, "--1") then
      T <= 1;
    else
      T <= 0;
    end if;
  end process;
end TestMatch;

```

**Figure 7.12.4 Application of Std\_Match Function in Package Numeric\_Std, (synths\lead7.vhd)**

**A** Avoid using “don’t cares” in synthesizable models.

*Rationale:* The use of “don’t cares” optimizes a design in terms of area utilization. However, there may be differences in the simulation results between a synthesized gate model with “don’t care” conditions and the original code. This is because the synthesizer may make use of intermediary product terms that are not in the original code. This difference in simulation outputs, between the two definition versions (original version and synthesized product), creates difficulties in the use of regression tests for design verification (see section 8.3). The area saving as the result of the use of “don’t cares” is relatively insignificant compared to the size of the design. However, the compromise in the application of automatic regression tests is significant, and does not justify the minimal savings in chip real estate.

### 7.13 PARAMETERIZED PRIORITY ENCODER

**Q** How can a priority encoder with up to 256 inputs be constructed?

**A** Priority (or First-one) encoders are used to encode an “n” input data stream into a binary number, and giving priority to the “first logical one” found in the input data.

A good application of such an encoder is the generation on an interrupt vector based upon “n” interrupt inputs. To maximize reusability, it is imperative that the model be parameterized. There are several methods to generate such an encoder. These schemes are a function of area and speed efficiency, and include the following:

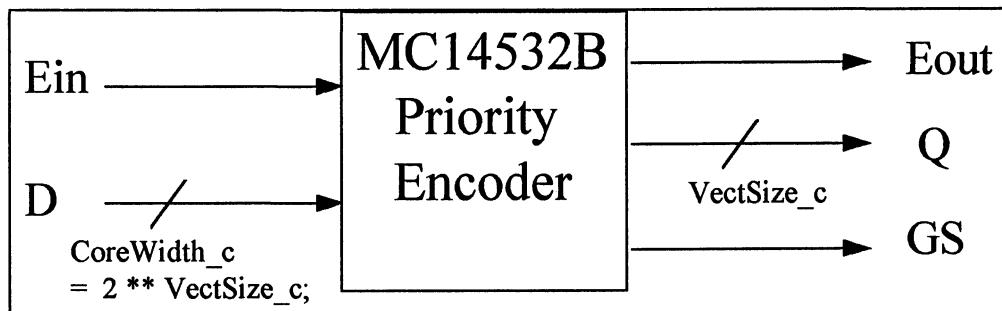
1. **Straight encoding.** This approach is inefficient in terms of speed when the number of inputs is large (e.g., greater than 16 or 32) because of many levels of combinatorial logic.

2. **Two level encoding.** This approach uses the straight encoding scheme for groups of inputs (e.g., 8 or 16 inputs per group) in the first level of encoding. The second level of encoding takes advantage of the results of the first encoding level to provide efficient hardware implementation. This enhances the overall speed because it reduces the overall levels of combinatorial logic.

These methods are discussed below.

### 7.13.1 Straight Encoding Priority Encoder

Figure 7.13.1-1 represents the design interface for a straight encoding priority encoder. Figure 7.13.1-2 represents a parameterized VHDL code that can be synthesized.



**Figure 7.13.1-1 Architecture of First One straight Encoding Design**

The model emulates the design of a commercial priority encoder, the *Motorola MC14532B*. That design represents a core component used in the creation of larger models through cascading. The core and the large model make use of a package (*Size\_Pkg*) to define constants parameters related to the following parameters: 1) number of inputs for the core and the larger design, 2) number of instances the core is instantiated in the larger design, and 3) some constants used in the model.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

package Size_Pkg is
  constant VectSize_c : Integer := 3; -- # of output bits
  constant CoreWidth_c : Integer := 2 ** VectSize_c; -- = 8
  constant DataSize_c : Integer := 32;
  constant Instances_c : Integer := DataSize_c / (CoreWidth_c); -- 32/8 = 4
  constant GS_c       : Std_Logic_Vector(CoreWidth_c - 1 downto Instances_c)
                      := (others => '0');
  constant VectorOut_c : Std_Logic_Vector(7 downto (2 * VectSize_c))
                      := (others => '0');
end Size_Pkg;
```

Annotations in the code:

- A callout box points to the line "constant CoreWidth\_c : Integer := 2 \*\* VectSize\_c; -- = 8" with the text "Used by core model".
- A callout box points to the line "constant Instances\_c : Integer := DataSize\_c / (CoreWidth\_c); -- 32/8 = 4" with the text "Used by large model".

```

-- Title      : Core Model
-- Description : Priority encoder
library IEEE;
use IEEE.Std_Language.all;
use IEEE.Std_Language.Arith.all;

library Work;
use Work.Size_Pkg.all;

entity MC14532B is
port(-- Data input
     D      : in  Std_Language_Vector(CoreWidth_c - 1 downto 0);
     -- Any of above device found an active bit, '0' if active
     Ein    : in  Std_Language;
     -- Any of data is asserted, or higher level has a bit asserted
     -- '0' if active
     Eout   : out  Std_Language;
     -- Vector out, active true
     Q      : out  Std_Language_Vector(VectSize_c - 1 downto 0);
     -- Device data is valid
     GS     : out  Std_Language -- MSB out
   );
end MC14532B;

architecture MC14532B_a of MC14532B is

begin -- MC14532B_a
-----  

-- Process: Priority_Lbl  

-----  

Priority_Lbl : process(D, Ein)
  variable Found1_v : Boolean;

begin -- process Priority_Lbl
  Eout <= '1'; -- initialized to prevent a latch
  Q <= (others => '0');
  GS <= '0';
  --
  if Ein = '1' then
    Search_Lbl : for I in D'range loop
      if D(I) = '1' then
        Eout <= '0';
        Q <= Std_Language_Vector(CONV_UNSIGNED(ARG => I,
                                                SIZE => VectSize_c));
        GS <= '1';
        exit Search_Lbl;
      end if;
    end loop Search_Lbl;
  else
    Eout <= '0'; -- to pass Q to lower devices
  end if;
end process Priority_Lbl;
end MC14532B_a;

```

**Figure 7.13.1-2 Parameterized Priority Encoder (synths\MC14532B.VHD)**

Some synthesizers do not yet implement the *exit* in a loop statement. Others do not implement the conversion function from integer to *Unsigned* type. For such synthesizers, figure 7.13.1-3 demonstrates a compilable code variation to the model described above. This model assumes that the number of inputs is eight.

```

architecture MC14532B_a of MC14532B is

begin -- MC14532B_a
-----
-- Process: Priority_Lbl
-----
Priority_Lbl : process(D, Ein)
variable Found1_v : Boolean;
type Lookup_Typ is array(7 downto 0) of
    Std_Logic_Vector(2 downto 0);
constant Lookup_c : Lookup_Typ
:= (7 => "111",
   6 => "110",
   5 => "101",
   4 => "100",
   3 => "011",
   2 => "010",
   1 => "001",
   0 => "000");

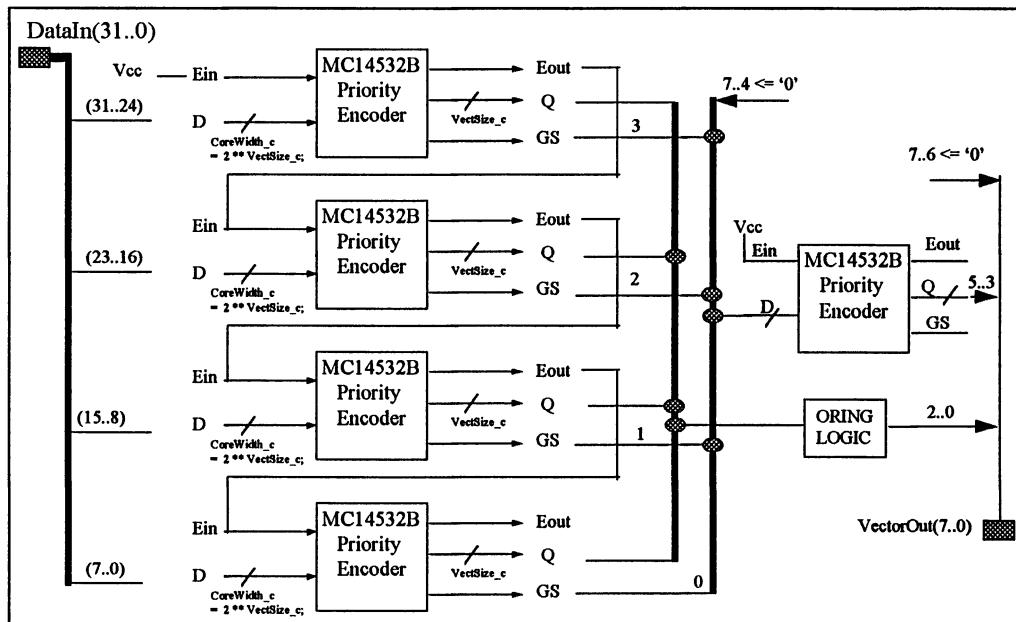
begin -- process Priority_Lbl
    Eout <= '1'; -- initialized to prevent a latch
    Q <= (others => '0');
    GS <= '0';
    Found1_v := False;
    --
    if Ein = '1' then
        Search_Lbl : for I in D'range loop
            if D(I) = '1' and not Found1_v then
                Eout <= '0';
                --Q <= Std_Logic_Vector(CONV_UNSIGNED(ARG => I,
                                                SIZE => 3));
                --
                Q <= Lookup_c(I);
                GS <= '1';
                Found1_v := True;
                -- exit Search_Lbl;
            end if;
        end loop Search_Lbl;
    else
        Eout <= '0'; -- to pass messagQ to lower devices
    end if;
end process Priority_Lbl;
end MC14532B_a;

```

**Figure 7.13.1-2 Parameterized Priority Encoder (MC14532q.VHD)**

### 7.13.2 Two level encoding

The two-level modeling of a priority encoder instantiates the cascadable MC14532B design multiple times. Figure 7.13.2-1 demonstrates the concept, and figure 7.13.2-2 provides the VHDL code.



**Figure 7.13.2-1 Two-Level Encoding Architecture for Priority Encoder**

```

library IEEE;
use IEEE.Std_Logic_1164.all;

library Work;
use Work.Size_Pkg.all;

entity Priority is
  port(DataIn    : in Std_Logic_Vector(DataSize_c - 1 downto 0);
       VectorOut  : out Std_Logic_Vector(7 downto 0) -- Vector signal
      );
end Priority;

architecture Priority_a of Priority is
  type Q_Typ is array(Instances_c - 1 downto 0) of
    Std_Logic_Vector(VectSize_c - 1 downto 0);

  component MC14532B
    port(
      -- Data input
      D      : in Std_Logic_Vector(CoreWidth_c - 1 downto 0);

      -- Any of above device found an active bit, '0' if active
      Ein   : in Std_Logic;

      -- Any of data is asserted, or higher level has a bit asserted
      -- '0' if active
      Eout  : out Std_Logic;

      -- Vector out, active true
      Q     : out Std_Logic_Vector(VectSize_c - 1 downto 0);

      -- Device data is valid
      GS    : out Std_Logic -- MSB out
     );
  end component;

```

This model can be used if the number of instances for the MC14532B is greater than one.

```

-- Used in 1st level encoding
signal Ein_s          : Std_Logic_Vector(Instances_c - 1 downto 0); -- 3 .. 0
signal Eout_s          : Std_Logic_Vector(Instances_c - 1 downto 0); -- 3 .. 0
signal GS_s            : Std_Logic_Vector(CoreWidth_c - 1 downto 0); -- 7 .. 0
signal Q_s             : Q_Typ;
signal VCC              : Std_Logic; -- + 5V
signal GND              : Std_Logic; -- Ground

-- Used for 2nd level encoding
signal VectorOut_s     : Std_Logic_Vector(7 downto 0);

begin

-----  

-- Ground/VCC  

-----  

GND <= '0';  

VCC <= '1';

-----  

-- GS_s bus initialization  

-- Purpose: Sets unused GS_s to 0, and unused VectorOut_s to 0  

-----  

-- ToZero_Lbl : for I in CoreWidth_c - 1 downto Instances_c loop  

--   GS_s(I) <= GND; -- ERROR, Drivers on ALL elements of array  

-- end loop ToZero_Lbl;  

-- ToZero2_Lbl : for I in VectorOut's left downto (2 * VectSize_c) loop  

--   VectorOut_s(I) <= GND;  

-- end loop ToZero2_Lbl; ↗  This code would produce multiple drivers (see 3.1)

-----  

-- Concurrent statements with no multiple drivers  

GS_s(CoreWidth_c - 1 downto Instances_c) <= GS_c; -- (others => '0'); --7..4  

VectorOut_s(VectorOut_c'range) <= VectorOut_c; -- (others => '0'); -- 7..6  

-----  

 Synopsys has a problem identifying the type of the right hand side  

with the (others => '0') because others is unconstrained. There  

should be no reason for this. Constant is used instead. See section 2.8.  

-- Most significant  

-- instance  

MC14532B_OI : MC14532B
port map (
    D      => DataIn((Instances_c - 1) * CoreWidth_c +      -- 3 * 8
                      CoreWidth_c - 1 downto                  -- 24 + 7  downto
                      (Instances_c - 1) * CoreWidth_c),      -- 31 .. 24
    Ein    => VCC,
    Eout   => Eout_s(Instances_c - 1), -- 3
    Q      => Q_s(Instances_c - 1), -- 3
    GS     => GS_s(Instances_c - 1) -- 3
);
-----  

-- Least significant instances
-----  

U: for K in Instances_c - 2 downto 0 generate -- 2 .. 0
    MC14532B_I : MC14532B
    port map (
        D      => DataIn((K * CoreWidth_c) + CoreWidth_c - 1 downto -- 23
                          (K * CoreWidth_c)), -- 16
        Ein    => Eout_s(K + 1),
        Eout   => Eout_s(K),
        Q      => Q_s(K),
        GS     => GS_s(K)
    );
end generate;

```

```

-- Process: Oring_Lbl
-- Purpose: Provides Oring of Lowee order priority decode bits

Oring_Lbl : process (Q_s)
  variable Or_v : Std_Logic_Vector(VectSize_c - 1 downto 0);
begin -- process Oring_Lbl
  Or_v := (others => '0'); -- Initialize and prevent a 'latch
  OrLp_Lbl : for I in Instances_c - 1 downto 0 loop
    Or_v := Q_s(I) or Or_v;
  end loop OrLp_Lbl;

  VectorOut_s(VectSize_c - 1 downto 0) <= Or_v;
end process Oring_Lbl;

-- 2nd level instance
MC14532B_2I : MC14532B
port map (
  D      => GS_s,
  Ein    => VCC,
  Eout   => open,
  Q      => VectorOut_s((2 * VectSize_c) - 1 downto VectSize_c), -- 5..3
  GS     => open
);

-- Vector Output
VectorOut <= VectorOut_s;
end Priority_a;

```

**Figure 7.13.2-2 Two-Level Encoding Architecture for Priority Encoder  
(Synths\Priority.vhd)**

Figure 7.13.2-3 represents a testbench for the priority encoder.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;

library Work;
use Work.Size_Pkg.all;
use Work.Image_Pkg;
use Work.LfsrStd_Pkg;

entity Priority_TB is
begin
end Priority_TB;

architecture Priority_TB_A of Priority_TB is
component Priority
  port(
    DataIn      : in Std_Logic_Vector(DataSize_c - 1 downto 0);
    VectorOut   : out Std_Logic_Vector(7 downto 0) -- Vector signal
  );
end component;

signal DataIn       : Std_Logic_Vector(DataSize_c - 1 downto 0)
                      := (others => '0');
signal VectorOut   : Std_Logic_Vector(7 downto 0);
signal Clk         : Std_Logic := '1';

```

```

begin
  Priority_I: Priority
  port map (
    DataIn      => DataIn,
    VectorOut   => VectorOut
  );

-----  

-- Clock generation  

-----  

Clk <= not Clk after 50 ns;

-----  

-- Process:  DataGen_Lbl  

-- Purpose:   Generate test vectors for DataIn input  

-----  

DataGen_Lbl : process
  variable Lfsr_v : Std_Logic_Vector(DataSize_c - 1 downto 0) :=  

    "1000101100101000110000101010100";
  variable Count_v : Unsigned(DataSize_c - 1 downto 0) :=  

    (others => '0');

  begin -- process DataGen_Lbl
    wait until Clk'event and Clk = '1';
    -- Lfsr_v := LfsrStd_Pkg.LFSR(Lfsr_v);  -- LFSR for random vectors
    -- DataIn <= Lfsr_v;
    Count_v := Count_v + 1;  -- Counter for sequential numbers
    DataIn <= Std_Logic_Vector(Count_v);
  end process DataGen_Lbl;

-----  

-- Process:  Verifier_Lbl  

-- Purpose:   Verifies that the output is the expected response  

-----  

Verifier_Lbl : process
  function Leading_One( s : Std_Logic_Vector ) return Integer is
    variable Leading_One_Position : Integer;
    variable S_v : Std_Logic_Vector(S'length - 1 downto 0);
  begin
    S_v := S;
    LP_Lbl: for I in S_v'range loop
      Leading_One_Position := I;
      if S_v(I) = '1' then
        exit LP_Lbl;
      end if;
    end loop;
    return Leading_One_Position;
  end;

  variable Expected_v : Std_Logic_Vector(7 downto 0);
begin -- process Verifier_Lbl
  wait until Clk'event and Clk = '1';
  Expected_v := Std_Logic_Vector(IEEE.NUMERIC_STD.TO_UNSIGNED
    (ARG  => Leading_One(DataIn),
     SIZE => 8));
  If VectorOut /= Expected_v then
    assert False
    report "DataIn = " & Image_Pkg.Image(DataIn) &
      "VectorOut = " & Image_Pkg.Image(VectorOut) &
      "Expected " & Image_Pkg.Image(Expected_v)
    severity Warning;
  end if;
end process Verifier_Lbl;
end Priority_TB_A;

```

Figure 7.13.2-3 Priority Encoder Testbench (synths\Prty\_TB.vhd)

## 7.14 GENERATING A SYNCHRONOUS PRECHARGE

**Q** | How can a precharge to a high logic level (i.e., Vcc) be designed for a synchronous signal that is about to be tri-stated?

**A** | A precharge is a technique to dynamically (with the push transfer gate) charge up an output to a logical ONE state prior to releasing the output to the tri-state level.

Figure 7.14-1 demonstrates a timing diagram of an ideal pre-charge where the *DataOut* signal is actively pumped up to a logical ONE during one half of a clock period following the output disable signal (*Enb*).

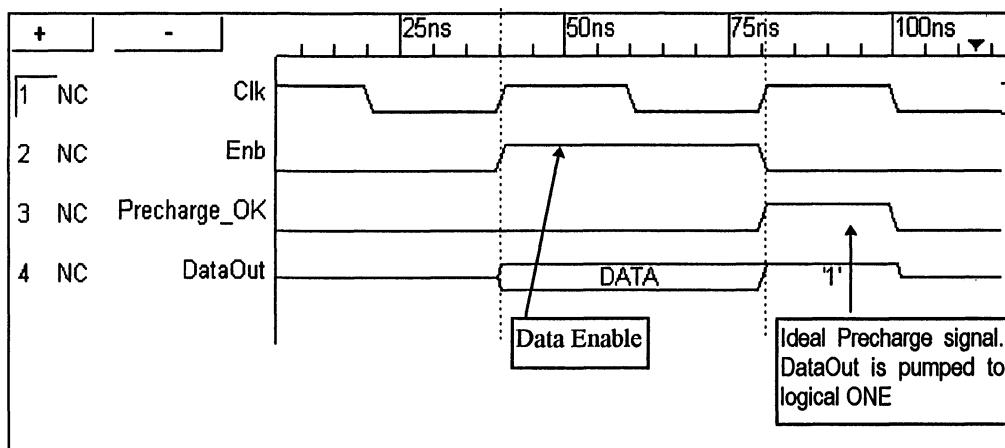


Figure 7.14-1 Timing Diagram of an Ideal Pre-charge

Figure 7.14-2 represents a logic schematic that implements a good precharge logic, and a logic with questionable performance.

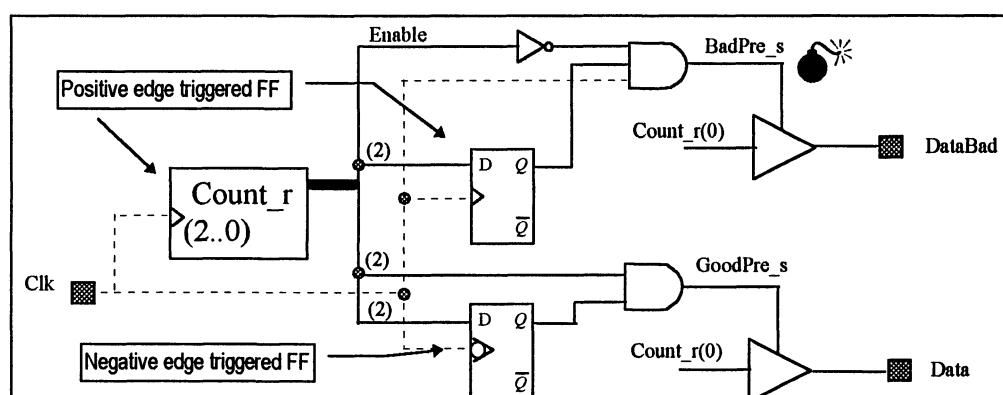


Figure 7.14-2 Precharge Logic Schematic

Figure 7.14-3 provides an interface timing analysis of the design using *Chronology's Quickbench Timing Analyzer* [17] tool.

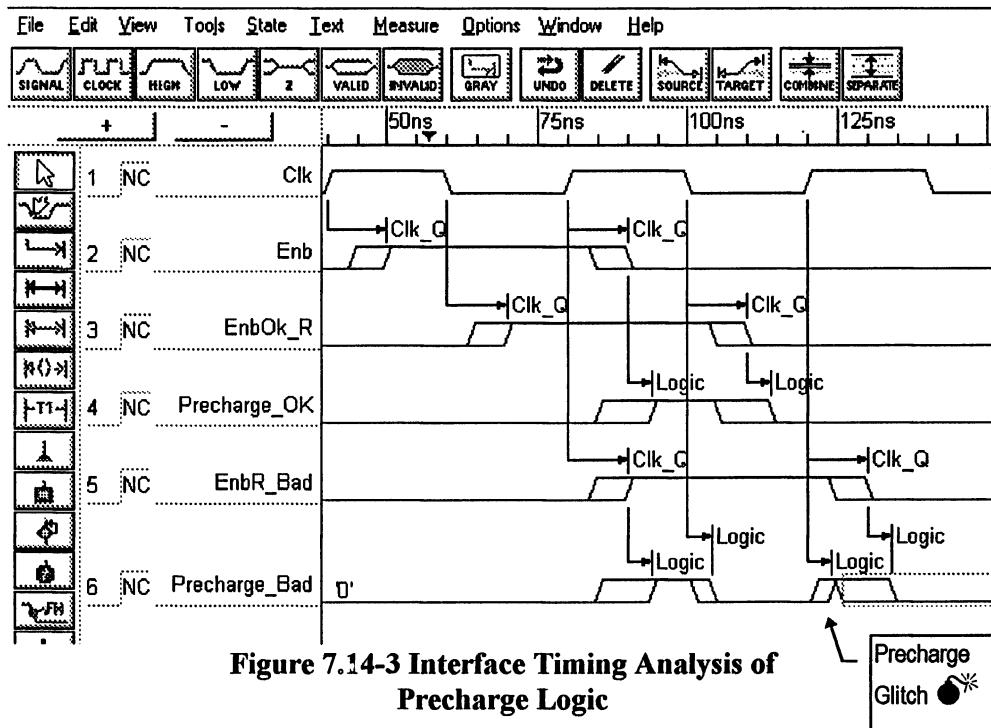


Figure 7.14-3 Interface Timing Analysis of Precharge Logic

Figure 7.14-4 represents the VHDL model for the precharge logic. Figure 7.14-5 demonstrates the potential precharge glitch using large propagation delays.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all; -- Synopsys package

entity Precharge is
  port (Clk      : in  Std_Logic;
        Reset    : in  Std_Logic;
        Data     : out Std_Logic;
        DataBad  : out Std_Logic);
end Precharge;

architecture Precharge_a of Precharge is
  signal Count_r   : Std_Logic_Vector(2 downto 0);
  signal Bad_r     : Std_Logic;
  signal Good_r    : Std_Logic;
  signal BadPre_s  : Std_Logic;
  signal GoodPre_s : Std_Logic;
```

```

begin -- Precharge_a
-----
-- Process: Counter_Lbl
-- Purpose: Generate test vectors
-----
Counter_Lbl : process
begin -- process Counter_Lbl
  wait until Clk'event and Clk = '1';
  if Reset = '1' then
    Count_r <= "000";
  else
    Count_r <= Count_r + 1;
  end if;
end process Counter_Lbl;

-----
-- Process: BadPrecharge_Lbl
-- Purpose: Demonstrates bad use of the precharge
-----
BadPrecharge_Lbl : process
begin -- process BadPrecharge_Lbl
  wait until Clk'event and Clk = '1';
  Bad_r <= Count_r(2);
end process BadPrecharge_Lbl;

-----
-- Process: GoodPrecharge
-- Purpose: Demonstrates good use of the precharge
-----
GoodPrecharge_Lbl : process
begin -- process GoodPrecharge_Lbl
  wait until Clk'event and Clk = '0';
  Good_r <= Count_r(2);
end process GoodPrecharge_Lbl;

-----
-- Tri-state Enables with precharge
-----
BadPre_s <= '1' when (Count_r(2) = '0') and
                (Bad_r = '1') and
                (Clk = '1') else
                '0';
GoodPre_s <= '1' when (Count_r(2) = '0') and
                (Good_r = '1') else
                '0';

-----
-- Data Out enable
-----
DataBad <= Count_r(0) when Count_r(2) = '1' else
               '1' when BadPre_s = '1' else
               'Z';

Data <= Count_r(0) when Count_r(2) = '1' else
               '1' when GoodPre_s = '1' else
               'Z';
end Precharge_a;

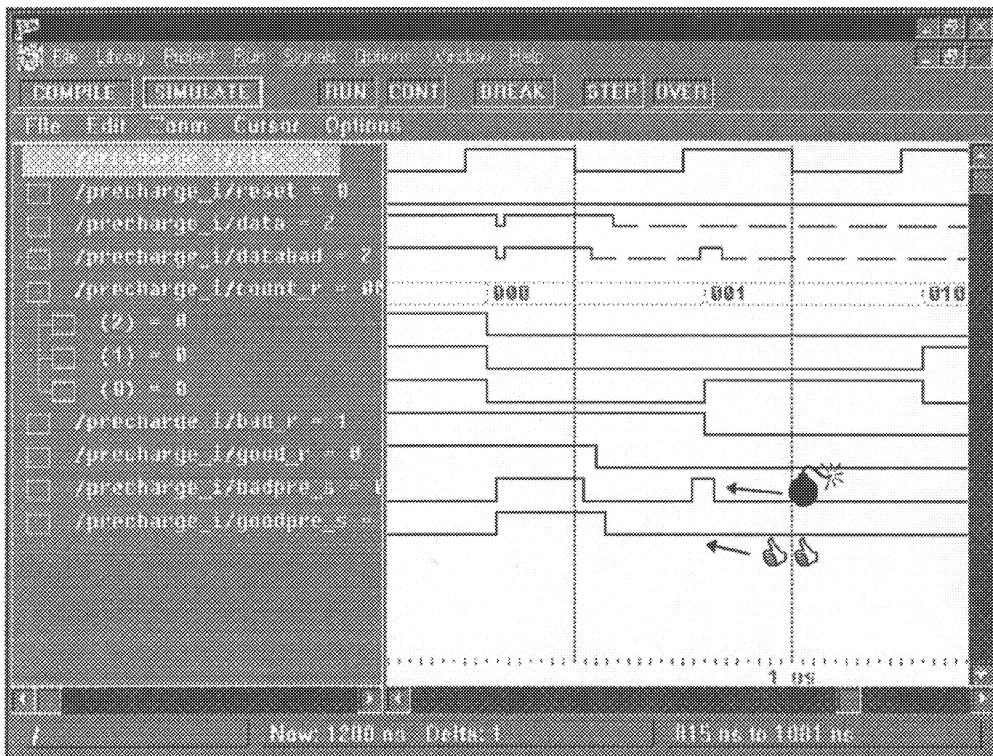
```

Count\_r(2) is the Tri-State enable signal.  
Count\_r(0) is the data sent to the tri-state buffer.

Tri-State enable signal is clocked with the negative edge of the clock. It is later used in the equation for the precharge.




Figure 7.14-4 Precharge Model (synths\prechrg.vhd)



**Figure 7.14-5 Simulation Results of Precharge Logic with Delays  
(synths\prech1.vhd) in a Testbench (synths\prechtb.vhd)**

## 7.15 TECHNOLOGY AND VHDL CODE DESIGN

**Q** | How does the technology impact the VHDL code design style?

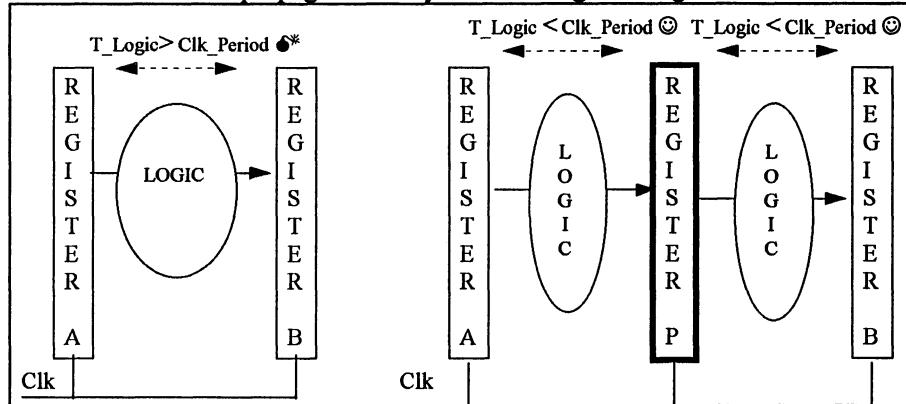
**A** | The VHDL code style is directed, no only by the hardware technology (e.g., Custom design, ASIC, FPGA, CPLD), but also by the synthesis tool technology and the libraries available for the targeted hardware technology. These factors include the following:

### 7.15.1 Basic architecture of a logic cell.

This is particularly true for Programmable Logic Devices (PLD), Complex Programmable Logic Devices (CPLD), and Field Programmable Gate Arrays (FPGA). These internal structures limit the amount and type of logic that can be implemented in a logic cell. To achieve complex logic implementations, many logic cells may be required in between register stages, thus impacting the propagation delays. The I/O buffer designs are other issues that deserve consideration because they impact, not only the drive characteristics, but also the propagation delays.

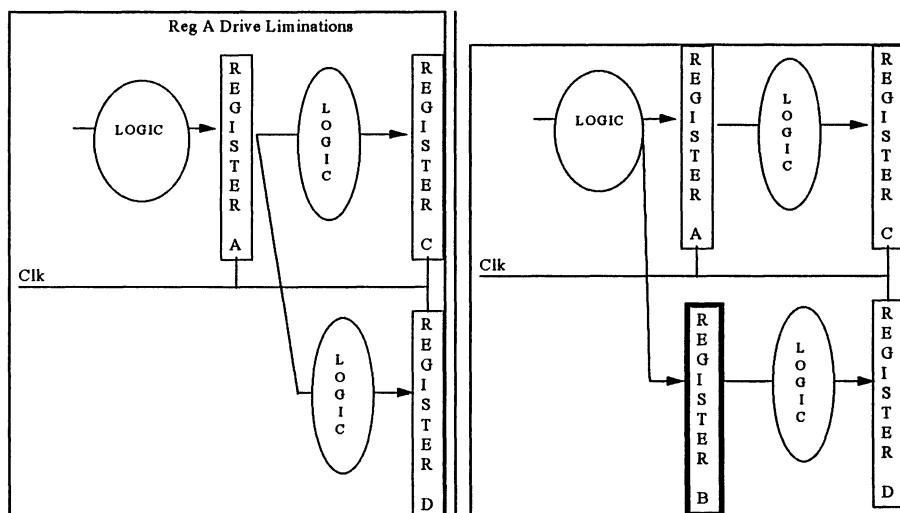
Common logic design techniques used to speed up designs are reflected in the VHDL code and include the following:

1. **Pipeline.** Pipelining adds registers in the combinational logic to meet the setup time. This is demonstrated in figure 7.15-1 where register P is the pipeline register used to minimize the propagation delays between register stages.



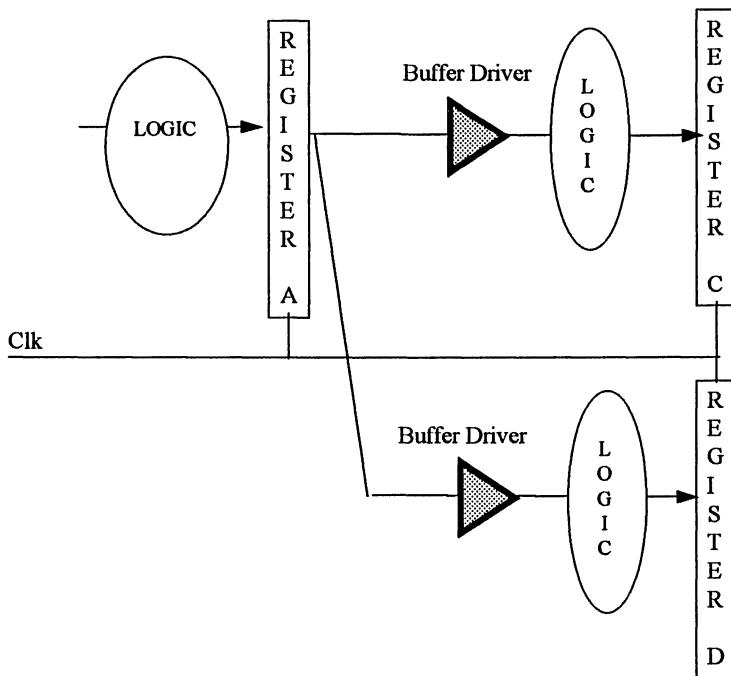
**Figure 7.15-1 Pipeline to Minimize Propagation Delays between Register Stages**

2. **Shadow Register.** Some situations arise where the output of a register is subjected to a heavy load, particularly if the driving logic is spaced far from the driving register, as shown in figure 7.15-2. This loading can cause excessive delays. To alleviate this problem, a shadow register (register B in the figure) can be used to hold a copy of the original register, and to split the original driving load.



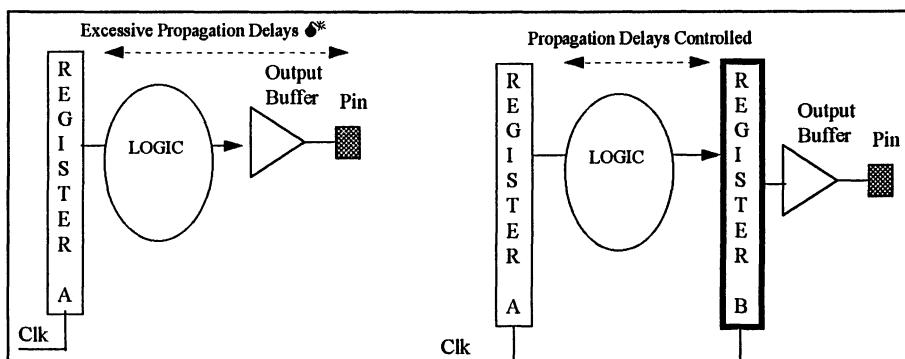
**Figure 7.15-2 Use of Shadow Registers to Minimize Loading Effects**

3. **Use of Buffers.** This approach is similar to the shadow register method, except that buffers are used to enhance the drive to selected logic clouds as shown in figure 7.15-3. Synthesizer tools tend to automatically provide buffering based on user defined parameters. To force individual drivers, one may need to define drivers at the structural level (through component instantiation).



**Figure 7.15-3 Use of Buffer Drivers to Minimize Loading Effects**

4. **Registered Outputs.** To guarantee minimum output delays (from the clock to the outputs) the outputs are often registered prior to being transferred to the I/O output buffers. This is demonstrated in figure 7.15-4.



**Figure 7.15-4 Registering Outputs to Minimize Clock to Output Delays**

### 7.15.2 Synthesis tool technology

The synthesis tools significantly impact the design performance, and conversely how VHDL must be coded, because logic minimization and mapping impact the internal propagation delays. If the synthesizer fails to achieve a design that meets the timing requirements, then the synthesis constraints and the architecture must be modified appropriately.

### 7.15.3 Supporting component libraries

Libraries can significantly impact the performance and the design time of a design. If tailored libraries components are available (e.g., ALU, Multiplier, Register file, FIFO, etc.) then the VHDL code takes a more structural approach with component instantiations. Otherwise, the VHDL code may take a more behavioral, RTL design approach.

## 7.16 SYNTHESIZING TRI-STATES



When the following code (figure 7.16-1) is synthesized, the synthesizer implements one set of tri-states and leaves the other set dangling. What is the problem? How can this code be corrected? What are the rules on using tri-states?

```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity TriState is
    port(Enable1 : in Std_Logic;
         Enable2 : in Std_Logic;
         In1Bus   : in Std_Logic_Vector(7 downto 0);
         In2Bus   : in Std_Logic_Vector(7 downto 0);
         OutBus  : out Std_Logic_Vector(7 downto 0));
end TriState;

architecture TriState_Err of TriState is
begin
    Why_Lbl: process(Enable1, Enable2, In1Bus, In2Bus)
    begin
        if Enable1 = '1' then
            OutBus <= In1Bus;
        else
            OutBus <= (others => 'Z'); ←
        end if;

        if Enable2 = '1' then
            OutBus <= In2Bus;
        else
            OutBus <= (others => 'Z'); ←
        end if;
    end process Why_Lbl;
end TriState_Err;

```

OutBus assigned HERE  
and  
HERE --

-- Last assignment WINS

**Figure 7.16-1 Incorrect Implementation of Tri-State (synths\trierr.vhd)**

**A** The problem is that statements in a process execute sequentially, and the delays are inertial. For each signal, the last signal assignment in a process prior to the execution of an explicit or implicit *wait* statement determines the final value of the signal, and all other assignments are canceled. In the above example, regardless of the value of *Enable1*, all the assignments within the *Enable1 if..else* clause are canceled, since they are reassigned within the *Enable2 if..else* clause. The goal is to make a single set of mutually exclusive signal assignments, as shown in figure 7.16-2. This code gives priority to *Enable1* in the event that both enable signals are active. Figure 7.16-3 demonstrates another alternative with the use of the *case* statement.

```
architecture TriState_OK of TriState is
begin
    Why_Lbl: process(Enable1, Enable2, In1Bus, In2Bus)
    begin
        if Enable1 = '1' then
            OutBus <= In1Bus;
        elsif Enable2 = '1' then
            OutBus <= In2Bus;
        else
            OutBus <= (others => 'Z');
        end if;
    end process Why_Lbl;
end TriState_OK;
```

Figure 7.16-2 Corrected Model of Tri-State with *if* (synths\triosk.vhd)

```
architecture TriState_OK2 of TriState is
begin
    Why_Lbl: process(Enable1, Enable2, In1Bus, In2Bus)
    begin
        case Std_Logic_Vector'(Enable1, Enable2) is
            when "10" | "11" => -- Enable1 = '1'           ← Use of an aggregate
                OutBus <= In1Bus;
            when "01" => -- Enable2 = '1'
                OutBus <= In2Bus;
            when others =>
                OutBus <= (others => 'Z');
        end case;
    end process Why_Lbl;
end TriState_OK2;
```

Figure 7.16-3 Corrected Model of Tri-State with *case* (synths\triosk2.vhd)

### 7.16.1 Latches in Tri-State Controls

[2] *Std\_logic\_1164 resolution function is synthesized into a three-state driver when a 'Z' is assigned onto a signal. In Synopsys® data type Std\_Logic cannot be used to model a wired-and or wired-or bus. Wired-and bussing scheme may be synthesized by writing a resolution function that contains the following comment:*

-- pragma resolution\_method wrired\_and

*It is also important to note that only one tri-state driver is synthesized per signal per process. This implementation is logical since one driver is created in a process for each signal where one or more signal assignments are made.*

In Synopsys®, when a signal (or a variable) is registered (or latched) in the same process in which it is tri-stated, then the enable control signal of the tri-state is also registered (latched). This concept is demonstrated in figure 7.16-4, and its synthesized implementation is shown in figure 7.16-5. This coding style may not be portable across synthesis products.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity DTri is
port(TriEnb : in Std_Logic;
      Din    : in Std_Logic;
      Clk    : in Std_Logic;
      Dout   : out Std_Logic);
end DTri;
architecture DTri_a of DTri is
begin
  DTri_Lbl: process(Clk)
  begin
    if Clk'event and Clk = '1' then
      if TriEnb = '1' then
        Dout <= 'Z';
      else
        Dout <= Din;
      end if;
    end if;
  end process DTri_Lbl;
end DTri_a;
```

Figure 7.16-4 Tri-State Enable Synchronous Control (synth\trisync.vhd)

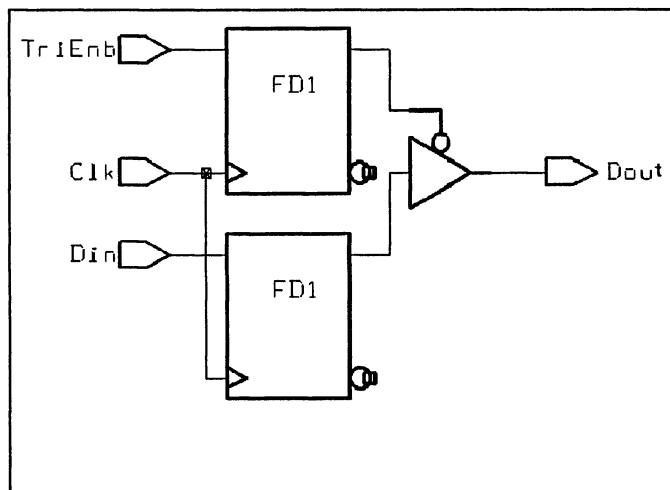


Figure 7.16-5 Tri-State Enable Synchronous Control Implementation

 To avoid the registering of the tri-state enable, define the tri-state output as a separate concurrent statement, as demonstrated in figure 7.16-6 and 7.16-7.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity DTri is
port(TriEnb : in Std_Logic;
      Din     : in Std_Logic;
      Clk     : in Std_Logic;
      Dout    : out Std_Logic);
end DTri;

architecture DTri_a of DTri is
signal TriData_s : Std_Logic;
begin
DTri_Lbl: process
begin
  wait until Clk'event and Clk = '1';
  TriData_s <= Din;
end process DTri_Lbl;

Dout <= TriData_s when TriEnb = '0' else 'Z';
end DTri_a;
```

Figure 7.16-6 Method to Avoid Registering of Tri-State (synths\trisync2.vhd)

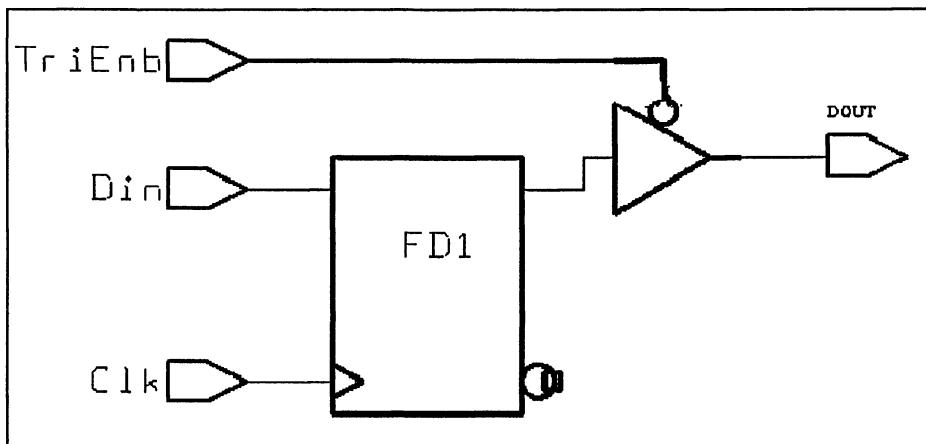


Figure 7.16-7 Implementation of Tri-State Code with No Implied Register

## 7.17 SUBELEMENT ASSOCIATION

**Q**

How can the concatenation of two or more signals of an architecture be passed to a parameter of a subprogram? Figure 7.17-1 demonstrates the problem. The concatenation operator or the subelement association method does not seem to work.

```
entity AssociationERR is
    port(Bus1 : in Bit;
         Bus9 : in Bit_Vector(8 downto 0);
         Bus10 : out Bit_Vector(9 downto 0));
end AssociationERR;

architecture AssociationERR_a of AssociationERR is
    procedure Invert(signal In_s : in Bit_Vector;
                     signal Out_s : out Bit_Vector) is
    begin
        Out_s <= not In_s;
    end Invert;

begin -- AssociationERR_a
    Test_Lbl : process(Bus1, Bus9)
    begin
        -- Following code is in error
        Invert(In_s => (Bus1 & Bus9),
               Out_s => Bus10); ←
        -- Following subelement association
        -- code seems non-portable
        Invert(In_s(9)          => Bus1,
               In_s(8 downto 0) => Bus9,
               Out_s            => Bus10); ←
    end process Test_Lbl;
end AssociationERR_a;
```

\* The concatenation is **ILLEGAL** because the "&" is an expression, and not a signal

This is legal in VHDL'93, questionable in VHDL'87 (see answer section). Generally non-synthesizable.

Figure 7.17-1 Subelement Association (synths\asoc\_err.vhd)

**A**

The concatenation of the actual parameters of a procedure call represents an expression and not a signal, and is illegal. For the 1987 standard, the issue of

subelement association is unclear, and it could be argued that this type of association is illegal. For 1993, it was the intent of the committee to clarify this, and allow sub-element association to define the subtype range for the formal. LRM'93 4.3.2.2.2 states: [1] a formal may be either an explicitly declared interface object or member 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 to the 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. This association rule applies to both port and subprogram associations. The association "in whole" (as shown in figure 7.15-2) is the most portable method among synthesizers and compilers. It utilizes a concurrent statement to compute the concatenation of the signals to be passed to

the interface.

```
-- ...
signal TenBit_s : Bit_Vector(9 downto 0);

begin -- AssociationOK_a
    TenBit_s <= Bus1 & Bus9;

    Test_Lbl : process(TenBit_s)
    begin
        Invert(In_s    => TenBit_s,
               Out_s   => Bus10);
    end process Test_Lbl;
end AssociationOK_a;
```

Association in whole. Signal `TenBit_s` is a concurrent statement that computes the concatenation of the 2 busses. That signal is passed as an actual into the procedure.

Figure 7.17-2 In-Whole Subelement Association (`synths\asoc_syn.vhd`)

Figure 7.17-3 demonstrates the concept of subelement association where the formal parameters are of constrained arrays. This code does not compile with some synthesizers because they do not handle subelement association.

```
entity AssociationOK is
port(Bus1 : in Bit;
      Bus9 : in Bit_Vector(8 downto 0);
      Bus10 : out Bit_Vector(9 downto 0));
end AssociationOK;

architecture AssociationOK_a of AssociationOK is
procedure Invert(signal In_s : in Bit_Vector(9 downto 0);
                 signal Out_s : out Bit_Vector(9 downto 0)) is
begin
    Out_s <= not In_s;
end Invert;

begin -- AssociationOK_a
    Test_Lbl : process(Bus1, Bus9)
    begin
        Invert(In_s(9)          => Bus1,
               In_s(8 downto 0) => Bus9,
               Out_s            => Bus10);
    end process Test_Lbl;
end AssociationOK_a;
```

Subelement of a interface object identify the formal with a locally static name.

Some synthesizers do not handle subelement associations.

Figure 7.17-3 Legal VHDL'93 Interface Association, but not Necessarily Synthesizable (`synth\asoc_sy2.vhd`)

### 7.17.1 Subelement Association for Ports

Figures 7.17.1-1 and 7.17.1-2 demonstrate the concept of subelement association for ports. The rules for port associations are the same as the ones described above. Port association "in whole" is the most portable and synthesizable.

```

entity InvertRol is
  port(A : in Bit_Vector;
       B : out Bit_Vector);
end InvertRol;

architecture InvertRol_a of InvertRol is
begin
... -- Architecture is irrelevant in this discussion
end InvertRol_a;

```

This model (as is) cannot be synthesized because the ports are unconstrained. Components instantiated in an architecture are synthesizable.

**Figure 7.17.1-1 Model of Component with Unconstrained interfaces  
(synths\ivtrol.vhd)**

```

entity InvertRol_Tb is
  port(A1 : in Bit;
       A2 : in Bit_Vector(14 downto 0);
       B : out Bit_Vector(15 downto 0));
end InvertRol_Tb;

architecture InvertRol_Tb_A of InvertRol_Tb is
  component InvertRol
    port(A : in Bit_Vector;
         B : out Bit_Vector);
  end component;
  signal T_s : Bit_Vector(15 downto 0);

begin
  T_s <= A1 & A2;

  U1_InvertRol: InvertRol -- 
    port map(
      A    => T_s,
      B    => B
    );
end InvertRol_Tb_A;

```

⌚ Association in whole. Signal T\_s is a concurrent statement which computes the concatenation of the 2 busses. That signal is passed as an actual into the procedure.

**Figure 7.17.1-2 Subelement Association of a Component (synths\ivtrltb.vhd)**

## 7.18 FINITE STATE MACHINE (FSM)

**Q** | What are the preferred coding styles for a FSM? What are the relevant issues?

**A** | There are two types of finite state machines:

- **Moore machine:** Where the outputs are a function of the current machine state (i.e., directly from a state register).
- **Mealy machine:** Where the outputs are a function of both the inputs and the current machine state (i.e., from state registers and combinational logic).

*There are three coding styles:*

1. *One process only handles both state transitions and outputs.*
2. *Two processes include a synchronous process for updating the state register, and a combinational process for conditionally deriving the next machine state and updating the outputs.*
3. *Three (or more) processes (and concurrent signal assignments) encompass a synchronous process for updating the state register, a combinational for conditionally deriving the next state machine, and a combinational process (or concurrent signal assignments) for conditionally deriving the outputs.*

Methods two and three represent the most straightforward approaches because they separate the synchronous registers from the combinational logic used in the computation of the next state and the outputs.

*One of the critical issues is the definition of the state encoding. By default, synthesis implements a state encoding that reflects the binary representation of the 'pos value of the machine's states enumeration type. Alternatively, the synthesis user may specify any of the following encoding styles:*

*One-hot, Gray, User-specified, Synopsys proprietary.*

**A**  when the default encoding is used, it is important to select the enumerations of the state machine such that the 'left' value represents the benign state. For example, if IDLE is the benign state, then it is recommended that the state type be defined as (IDLE, BUSY, WAITING, ...) rather than (BUSY, IDLE, ...).

*Rationale: The default will typically be implemented when the state register equals zero. This typically represents the RESET state.*

## 7.19 ONE-HOT ENCODING

**Q** | How can a “one-hot” encoding be specified?

**A** | “One-hot” encoding means that the state machine is encoded such that each bit of the state register defines a state. Thus, the following could be used.

```
constant S0 : Std_Logic_Vector(3 downto 0) := "0001";
constant S1 : Std_Logic_Vector(3 downto 0) := "0010";
constant S2 : Std_Logic_Vector(3 downto 0) := "0100";
constant S3 : Std_Logic_Vector(3 downto 0) := "1000";
```

However, if a *case* statement is used for the state transitions, then a synthesizer will decode the entire state vector. This is not “one-hot” encoding, but rather one of many different ways that the state vectors can be assigned. For example:

```
case State_Vector is
    when S0 => ....
    when S1 => ....
end case;
```

Not “one-hot” encoding because all bits of state vectors are decoded.

In Synopsys®, it is possible to use a two-pass synthesis approach, along with proper directives, to derive a state-vector that is decoded on a per-bit basis (i.e., “one-hot”), and not on a per-vector basis.

It is possible to write the code such that true “one-hot” encoding will be synthesized as shown in figure 7.19. This is a one pass synthesis process, and is more portable.

```
process
begin
    wait until Sysclk'event and Sysclk='1';
    if Nreset = '0' then
        state_vector <= "00001";
    else
        -- Action for S0 --
        if state_vector(0) = '1' then
            ..
        end if;
        --
        if state_vector(1) = '1' then
            ..
        end if;
    end process;
```

True one-hot encoding because each bit defines a state of the state machine.

**Figure 7.19 Manual Emulation of One-Hot Encoding**

## 7.20 INSTANTIATING SYNOPSYS® DESIGNWARE COMPONENTS

**Q** What are Synopsys® DesignWare components and how can the components be instantiated?

**A** DesignWare represents a library of components that are optimized for area and/or speed. For example, figure 7.20 represents the instantiation of a RAM.

```
-- Declarations
library DWARE, DW03;
use DWARE.DWpackages.all;
use DW03.DW03.components.all;

-- Instantiations
U0: DW03_ram2_a_d  -- dual-port RAM
generic map(Depth      => Desired_Depth,
            Data_Width => Desired_Width)
port map(DataIn   => Input_Data_Lines,
         WAddr     => Write_address_Lines,
         RAddr     => Read_Address_Lines,
         CSB       => RAM_Select_Line,
         WRB       => Write_Enable_Line,
         DataOut  => Data_Out_Lines);
```

**Figure 7.20 Instantiation of a RAM component with DesignWare**

## 7.21 RESOURCE SHARING

**Q** | What is “resource sharing” and how can “resource sharing” be used to control the design?

**A** | “Resource sharing” is the design approach to share expensive hardware resources among several potential users of the resource. Examples of expensive resources that are generally shared include: multipliers, adders, and comparators. There are three methods to achieve resource sharing<sup>14</sup> with Synopsys®:

- **Automatic:** The compiler evaluates the VHDL description and makes resource sharing decisions to optimize the design to meet the constraints.
- **Automatic with manual controls:** Manual control statements in the VHDL code are used to assign operations to resources explicitly.
- **Manual:** Manual controls are used to assign operations to resources, and to share operations explicitly assigned to the same resource.

The reader should use the referenced Synopsys® manual for more details on the Synopsys® specific methods. However, some general guidelines are provided below.

- Operations can be shared only if they are in the same process.
- Resource sharing algorithm assumes that all if statements are independent. Consequently, the VHDL model shown in figure 7.21-1 will result in two adders, even though the if conditions are mutually exclusive. Using a single if..elsif...else statement allows resource sharing if it is enabled, as shown in figure 7.21-2.

```
if Add_AB = '1' then
    Out_c <= A + B;
end if;

if AB /= '1' then
    Out_c <= E + F;
end if;
```

Independent if statements, thus  
 NO resource sharing

Figure 7.21-1 No Resource Sharing is Implied -- Two Adders are Synthesized

```
library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.std_logic_arith.all;
entity Sharing is
    port (A : in  Std_Lock_Vector(3 downto 0);
          B : in  Std_Lock_Vector(3 downto 0);
          C : in  Std_Lock_Vector(3 downto 0);
          D : in  Std_Lock_Vector(3 downto 0);
          E : in  Std_Lock;
          Q : out Std_Lock_Vector(3 downto 0));
end Sharing;
```

```

architecture sharing_a of Sharing is
begin -- sharing_a
  Sharing_Lbl : process (A, B, C, D, E)
    begin -- process Sharing_Lbl
      if E = '1' then
        Q <= UNSIGNED(A) + UNSIGNED(B);
      else
        Q <= UNSIGNED(C) + UNSIGNED(D);
      end if;
    end process Sharing_Lbl;
end sharing_a;

```

Single *if* statement, thus resource sharing is implied (if it is enabled)

**Figure 7.21-2 Automatic Resource Sharing (synths\sharing.vhd)**

- Resource sharing can be forced through lower level of abstractions. For example, multiplexers can be explicitly defined in the architecture, with the outputs of those multiplexers feeding a single adder. This is demonstrated in figure 7.21-3.

```

architecture sharing_a of Sharing is
  signal AddIn1_s : Std_Logic_Vector(3 downto 0);
  signal AddIn2_s : Std_Logic_Vector(3 downto 0);
begin -- sharing_a
  AddIn1_s <= A when E = '1' else C; -- Mux
  AddIn2_s <= B when E = '1' else D; -- Mux
  Q <= Unsigned(AddIn1_s) + Unsigned(AddIn2_s);
end sharing_a;

```

**Figure 7.21-3 Resource Sharing by Design (synths\sharingf.vhd)**

## 7.22 APPLYING DIGITAL DESIGN EXPERIENCES

**Q**

How can the code in figure 7.22-1 be optimized to reduce area size?

```

entity AreaOpt is
  port (D : in Bit_Vector(7 downto 0); -- data
        E : in Bit; -- Enable
        IA : in Integer range 0 to 7; -- index
        IB : in Integer range 0 to 7; -- index
        K : out Bit; -- Output
        Q : out Bit); -- Output
end AreaOpt;

architecture AreaOpt_Least of AreaOpt is
begin -- AreaOpt_Least
  Area_lbl : process (D, E, IA, IB)
    begin -- process Area_lbl
      K <= '0';
      Q <= '0';
      if E = '1' then
        K <= D(IA);
      else
        Q <= D(IB);
      end if;
    end process Area_lbl;
  end AreaOpt_Least;

```

2 Multiplexers implied by synthesis

**Figure 7.22-1 Model for Optimization (synths\areaop1.vhd)**

**A** This example demonstrates how an understanding of the logic can optimize a design. In this case, to reduce area it is important to eliminate unnecessary decode.

Figure 7.22-3 demonstrates the concept. In this case, it is necessary to compute the decoding index and the data to be sent out first. The routing of the data is computed afterward. This approach is more efficient in area, but is slower in speed. Based on area and critical path numbers given at the Synopsys Users Group presentation, figure 7.22-2 provides comparative results. In addition, compilation results using QuickLogic® FPGA and Synplify® synthesizer are also included for comparison.

TOOL	Area (relative)	Critical Path (ns)
Synopsys® -- AreaOp1	83	5.71
Synopsys® -- AreaOp2	46	7.46
Simplify® -- AreaOp1	6 out of 96 cells (6.2%)	13.9
Simplify® -- AreaOp2	5 out of 96 cells (5.2%)	21.2

**Figure 7.22-2 Comparative Area After Synthesis**

```

architecture AreaOpt_Most of AreaOpt is
begin -- AreaOpt_Least
  Area_lbl : process (D, E, IA, IB)
    variable I_v : Integer range 0 to 7;
    variable Out_v : Bit;
  begin -- process Area_lbl
    if E = '1' then
      I_v := IA;      -- index definition
    else
      I_v := IB;
    end if;

    Out_v := D(I_v);  -- optimization. Output value is defined.
                        -- Must now determine where to send it
    K <= '0';
    Q <= '0';
    if E = '1' then
      K <= Out_v;
    else
      Q <= Out_V;
    end if;
  end process Area_lbl;
end AreaOpt_Most;

```

**Figure 7.22-3 Optimized Model for Area (synths\areaop2.vhd)**

## 7.23 ADDRESS RANGE IDENTIFICATION VIA INFERRED COMPARATOR

**Q** Given the following model in figure 7.23-1, how can the code be modified to improve its performance?

```
entity Comparator is
  port (A : in Bit_Vector(31 downto 20);
        Q : out Bit);
end Comparator;

architecture Comparator_Cmplx of Comparator is
begin -- Comparator_Cmplx
  Comparare_Lbl : process (A)
    -- Inequality operator infers 2 12-bit comparators
  begin -- process Comparare_Lbl
    if ((A <= "000000000110") and
        (A >= "000000000001")) then
      Q <= '1';
    else
      Q <= '0';
    end if;
  end process Comparare_Lbl;
end Comparator_Cmplx;
```

← Comparators implied by the  
 "<=" and ">=" operators.

Figure 7.23-1 Model of a Comparator (synths\compart1.vhd)

**A** The inequality operators, though appropriate for simulation, will infer the use of comparators. It would be more efficient to search for a pattern in the address ranges that will circumvent the need for the usage of the comparator operator. This approach is demonstrated in figure 7.23-2.

```
architecture Comparator_Better of Comparator is
begin -- Comparator_Cmplx
  Comparare_Lbl : process (A)
    -- Code infers one 9-bit AND
    --           two 3-bit NANDS
    --           one 3-bit AND

  begin -- process Comparare_Lbl
    if ((A(31 downto 23) = "000000000") and
        (A(22 downto 20) /= "111") and
        (A(22 downto 20) /= "000")) then
      Q <= '1';
    else
      Q <= '0';
    end if;
  end process Comparare_Lbl;
end Comparator_Better;
```

← Simple logic implied by the  
 "=" and "/=" operators.

Figure 7.23-2 Address Range Identification without Inference of Comparators (synths\compart2.vhd)

## 7.24 PORT MAPPING TO GROUND OR VCC

**Q**

How can a port of an instantiated component be port mapped to Ground or VCC?  
The following code does not work.

U1\_XYZ: XYZ -- component instantiation

```
port map(A => '0', -- input to ground
          B => '1', -- input to VCC
          C => C, -- input to C
          D => open, -- output to open
          Q => Q); -- output to Q
```

Legal in VHDL'93.  
Illegal in VHDL'87 and  
in synthesis •\*

**A**

The actual in a port map association to an input port must be a signal in synthesis. In VHDL, but not in synthesis, it could be associated with an *open*, only if the input port is initialized. It could be associated with the *open* when the port is an *inout* or *out* port. In synthesis, a ground and VCC signal must be declared, and a concurrent assignment must be made to those signals. The port element associations can be made to those signals as shown in Figure 7.24-1.

```
architecture ToGndVCC_a of ToGndVCC is
  component XYZ
    port(
      A : in      Bit;
      B : in      Bit;
      C : in      Bit;
      D : out     Bit;
      Q : out     Bit);
  end component;

  signal Gnd   : Bit;
  signal VCC   : Bit;
  signal C     : Bit;
  signal Q     : Bit;
begin -- ToGndVCC_a
  Gnd <= '0'; -- Assign value to Ground signal
  VCC <= '1'; -- Assign value to VCC signal

  U1_XYZ: XYZ -- component instantiation
    port map(A => Gnd, -- input to ground
              B => Vcc, -- input to VCC
              C => C, -- input to C
              D => open, -- output to open
              Q => Q); -- output to Q
end ToGndVCC_a;
```

Signal initialization is not allowed in synthesis

**Figure 7.24-1 Element Association to Ground, Vcc, and Open  
(synths\ToGndVcc.vhd)**

## 7.25 BIT REVERSAL

**Q** | How can a bit reversal operation be defined?

**A** | The code shown in figure 7.25-1 provides an example of a bit reversal process. Figure 7.25-2 uses a bit reversal function.

```
entity BitReversal is
  generic (Size_g : Integer := 4);
  port (A : in Bit_Vector(Size_g - 1 downto 0);
        Q : out Bit_Vector(Size_g - 1 downto 0));
end BitReversal;

architecture BitReversal_aP of BitReversal is
begin -- BitReversal_aP
  BitRev_Lbl : process (A)
    variable A_v : Bit_Vector(Size_g - 1 downto 0);
    variable B_v : Bit_Vector(Size_g - 1 downto 0);
  begin -- process BitRev_Lbl
    B_v := A; -- Variable copies data from port A

    Lp_Lbl : for K in Size_g - 1 downto 0 loop
      A_v(K) := B_v(Size_g - 1 - K);
    end loop Lp_Lbl;
    Q <= A_v;
  end process BitRev_Lbl;
end BitReversal_aP;
```

Normalization used to control range and direction

Use of variable assignment instead of signal assignment insures proper bit selection (because of normalization). It also enhances simulation speed (over signal assignment alternative).

Figure 7.25-1 Bit Reversal Process (synths\bitrevp.vhd)

```
architecture BitReversal_a of BitReversal is

  function BitReverse(A : Bit_Vector) return Bit_Vector is
    variable A_v : Bit_Vector(A'length - 1 downto 0);
    variable B_v : Bit_Vector(A'length - 1 downto 0);
  begin -- process BitRev_Lbl
    A_v := A;
    B_v := A;

    Lp_Lbl : for K in A_v'range loop
      A_v(K) := B_v(A_v'length - 1 - K);
    end loop Lp_Lbl;
    return A_v;
  end BitReverse;

  begin -- BitReversal_a
    Q <= BitReverse(A);
  end BitReversal_a;
```

Normalization used to control range and direction

Figure 7.25-2 Bit Reversal with Function (synths\bitrevs.vhd)

## 7.26 HOW TO DESIGN A TIMER IN VHDL

**Q** | What techniques can be used to design a synthesizable up or down timer?

**A** | The issue in designing a timer is how the rollover is handled. There are basically four methods:

1. Use of the *if* statement to test for the rollover condition
2. Use of the *case* statement to test for the rollover condition
3. Use of the *mod* operator.
4. Use the overloaded “+” with type *Unsigned* and integer. This “+” operator limits the results to the size of parameters (i.e., the carry is discarded)

Methods 1 and 3 are demonstrated in figure 7.26-1. Method 3 is also shown in section 1.5.3. Figure 7.26-2 demonstrates the use of method 4. Method 2 is shown in section 9.2.2..

```
entity Timer is
  port(ResetF : in  Bit;
        Clk   : in  Bit;
        Timer1 : out Integer range 0 to 7;
        Timer2 : out Integer range 0 to 7);
end Timer;

architecture Timer_a of Timer is
  constant TimerSize_c : Integer := 8;
  constant MaxCount_c : Integer := TimerSize_c - 1;
  signal  Timer1_s    : Integer range 0 to MaxCount_c;
  signal  Timer2_s    : Integer range 0 to MaxCount_c;

begin
  Timer1_Lbl: process
  begin
    wait until Clk'event and Clk = '1';
    if (ResetF = '0') then
      Timer1_s <= 0;
    elsif (Timer1_s = MaxCount_c) then
      Timer1_s <= 0 ;
    else
      Timer1_s <= Timer1_s + 1;
    end if;
  end process Timer1_Lbl;

  Timer2_Lbl: process
  begin
    wait until Clk'event and Clk = '1';
    if (ResetF = '0') then
      Timer2_s <= 0 ;
    else
      Timer2_s <= (Timer2_s + 1) mod TimerSize_c;
    end if;
  end process Timer2_Lbl;

  Timer1 <= Timer1_s;      -- Concurrent statements
  Timer2 <= Timer2_s;      -- for assignments to Timer outputs
end Timer_a;
```

**Figure 7.26-1 Synthesizable Timer Design -- Integer Methods  
(synths\timer\_ea.vhd)**

```

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.std_logic_arith.all;

entity Timer is
    -- 3-bit counter (Count) which synchronously is reset to
    -- "000" when Reset = '1'.
    port(Reset      : in Std_Logic;
          Clk        : in Std_Logic;
          Data       : out Std_Lock_Vector(2 downto 0));
end Timer;

architecture Timer_a of Timer is
    signal Count_s : Std_Lock_Vector(2 downto 0);
begin
    Count_lbl: process
    begin
        wait until Clk'event and Clk = '1';
        if Reset = '1' then
            Count_s <= "000";
        else
            Count_s <= Unsigned(Count_s) - 1;
        end if;
    end process Count_Lbl;

    -- Concurrent signal assignment
    Data <= Count_s;
end Timer_a;

```

Use "+" for up-counter,  
"-" for a down-counter  
Type conversion to *Unsigned*. This  
overloaded "+" operator limits results  
size of parameters

**Figure 7.26-2 Synthesizable Timer Design -- Unsigned Method  
(synths\timr3\_ea.vhd)**

## 7.27 SPECIFYING A MULTIPLIER

**Q**

How can a multiplier circuit be defined in synthesis?

**A**

There are various multiplier architectures. The most efficient multipliers would most likely be one from a library (such as DesignWare) because it is optimized for speed or area. Another option is to use the multiply operator in the RTL/behavioral VHDL code, and let the synthesis tool determine the architecture, based on the constraints. Figure 7.27 demonstrates the use of the multiplier operator with the *Std\_Lock\_Arith* package.

```

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.Std_Lock_arith.all;
entity Multiply is
    port(A : in Std_Lock_Vector(3 downto 0);
          B : in Std_Lock_Vector(3 downto 0);
          C : out Std_Lock_Vector(7 downto 0));
end Multiply;

```

```

architecture Multiply_a of Multiply is
begin
  Mpy_Lbl: process(A, B)
    variable C_v : Unsigned(7 downto 0);
  begin
    C_v := Unsigned(A) * Unsigned(B);
    C <= Std_Logic_Vector(C_v);
  end process Mpy_Lbl;
end Multiply_a;

```

**Figure 7.27 Use of the Multiplier Operator (synths\multpy.vhd)**

## 7.28 BEHAVIORAL SYNTHESIS

**Q** | What is Synopsys® behavioral synthesis?

**A** | Behavioral synthesis allows the writing of synthesizable models at a higher level of abstraction. The overall architectural dataflow and control options may automatically be experimented with, and do not have to be laboriously entered manually via a lengthy and explicit VHDL description. Behavioral compiler allows:

- **VHDL after clause.** Delay time must be a multiple of the clock period.
- **Arrays of arrays may be mapped to a RAM.** All references to this array are mapped onto reads and writes of the corresponding RAM.
- **Combinational operations may automatically be extended over multiple cycles.** Multiple cycles can be inferred through behavioral code, and resource sharing operations are allowed on these parts.
- **Finite state machines may be automatically inferred.**

Consider the code segment shown in figure 7.28 to implement the multiplication of two complex numbers:

```

wait until Clock = '1';
A := Input_Port1;
wait until Clock = '1';
B := Input_Port_2;
wait until Clock = '1';
C := Input_Port_3;
wait until Clock = '1';
D := Input_Port_4;
wait until Clock = '1';
CR <= A * C - B * d;
CI <= A * D + B * C;
wait until Clock = '1';

```

**Figure 7.28 Behavioral Compiler Sample Code**

The behavioral compiler can automatically do architectural trade-offs. This is based on the simple observation that the computations do not have to wait for the arrival of all the operands. In using

*behavioral compiler, the following methodology is used:*

- *Designer specifies clock cycles (latency).*
- *Behavioral compiler specifies resources, registers, multiplexers, finite state machines, dataflow/control.*
- *Behavioral compiler gets area and delays estimates for resources using DesignWare Library.*

The reader should refer to Synopsys® documentation on the subject of behavioral synthesis.

## **8. DESIGN VERIFICATION AND TESTBENCH**

---

---

Design verification is the process of insuring design correctness. Verification typically encompasses three classes of disciplines:

1. **Functional verification** during the architectural/RTL design.
2. **Regression tests** for modified designs as a result of synthesis, design optimization, post-layout, etc.
3. **Formal verification** during design validation, synthesis, or optimization.

These methods are discussed in this section, along with other topics related to testbench designs. Most of the ideas on functional verification were contributed by the author<sup>[9]</sup>. Regression tests methodologies were contributed by Sandi Habinc<sup>[2]</sup> and Peter Sinander (from the European Space Agency). Formal verification methodologies were contributed by Compass Design Automation.

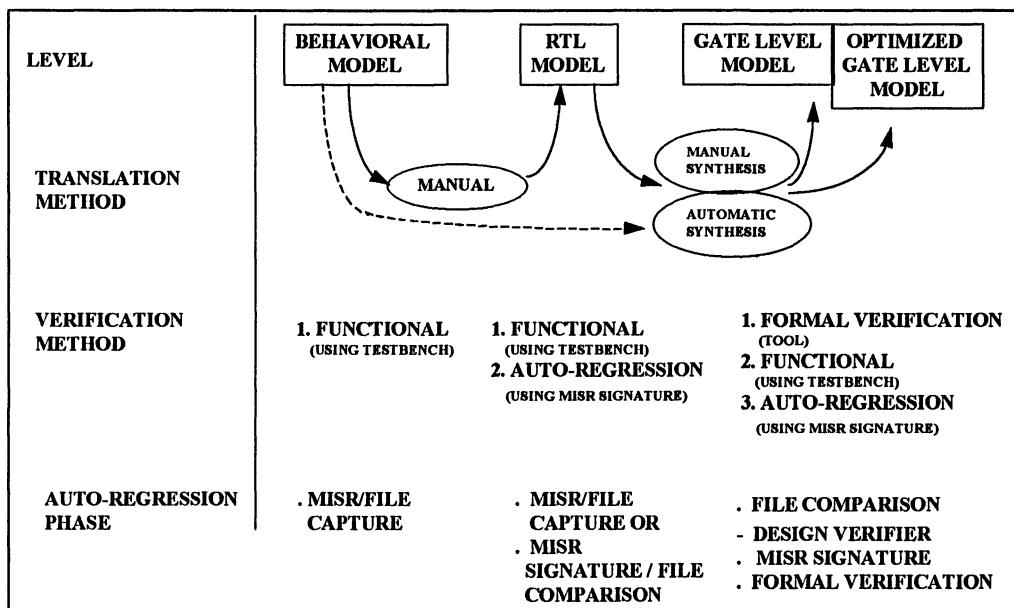
Other requirements in the field of verification encompass the hardware and software interactions in a virtual prototyping environment, timing verification, and intelligent formal verification processes. Those topics are not covered in this chapter because they are part of an evolving technology that is tool specific. The reader is encouraged to explore these new technologies with tool vendors.

## 8.1 VERIFICATION PROCESSES

**Q** In a lifetime of a design, what are the verification processes?

**A** A design typically consists of several phases, depending upon the methodologies and tools used in the design process. Since these vary significantly among users, a template skeleton is presented here for the purpose of discussions. It does not endorse any specific methodology. Figure 8.1 represents the life cycle of a design. It includes the following:

1. Levels of designs,
2. Typical translation methods between design levels,
3. Typical verification methods at each of design level,
4. Auto-regression techniques to validate, at each level of translation, that the functionality of the design has not changed.



**Figure 8.1 Design Levels, Translation tools, and Verification Methods**

If the behavioral level is the highest level of design entry, then verification consists of functional testing (typically in a testbench, to insure that the design functions as intended). The testbench includes the Unit Under Test (UUT) and the environment around the UUT (see section 8.2). During this stage, the UUT is subjected to a set of stimulus vectors that emulate the UUT's environment. The values of the UUT's stimulus and response signals can be captured either in files or in a Multiple Input Signature Register (MISR) (see sections 5.3 and 8.6).

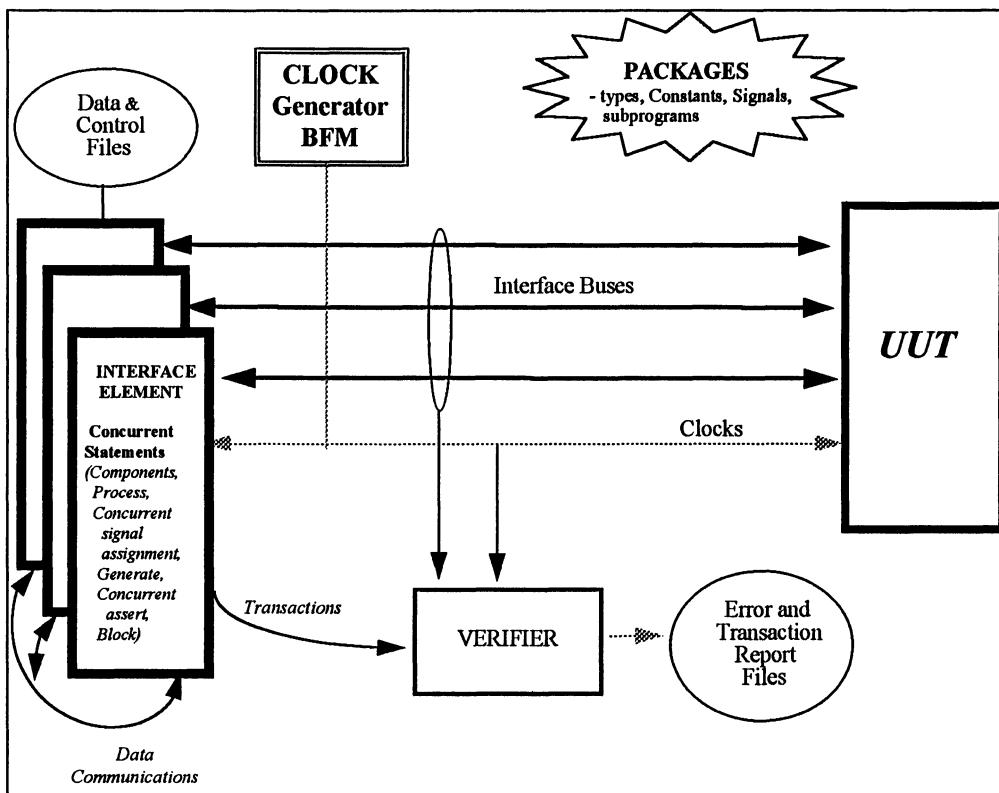
As the design is decomposed to lower levels of abstractions, verification of a lower level design can be performed using the same techniques as the ones used at the higher levels of the design. Verification can also be done by taking advantage of the results achieved at a higher level of abstraction. These methods are called “auto-regression” and include either signature analysis using a MISR, or the use file comparisons between simulation runs. Another alternative to auto-regression is the use of formal verification, discussed in section 8.4.

## 8.2 FUNCTIONAL VERIFICATION

**Q** | What are the functional verification techniques?

**A** | Functional verification verifies that the UUT model corresponds to the UUT specifications when subjected to a specified set of test vectors. This verification can be performed either manually or automatically. **Manual verification** is very tedious and is subject to error because it requires visual examinations and analysis of the trace signals. **Automatic verification** involves the use of verifiers (or checkers) that dynamically (during simulation) compare the operation of the UUT with the expected results. The verifier is a VHDL model that examines the stimulus and response signals connected to the UUT, and detects violations. The verifier is typically a different representation of the UUT that is used in the verification process. To detect any violation (or design error), the verifier uses its knowledge of the UUT’s operation, the environment stimulating the device, information about the UUT’s internal signals (see section 9.2), and the UUT’s interface responses to the external stimuli. The verifier can also be used as a smart monitor (or logic analyzer) to store, in a human readable format, the pertinent UUT’s transactions. That information is often useful for documentation and debugging.

Figure 8.2 represents a generic testbench architecture that incorporates the UUT, the Bus Functional Models (BFM), and other VHDL constructs. BFM design methodologies are addressed in section 8.6. A complete example of a UUT design (a Universal Asynchronous Receiver Transmitter or UART), testbench, and verifier can be found in the referenced book<sup>[9]</sup>. In that example, the verifier is checking that the character transmitted by a UART is properly received by a receiver. The message is sent over a transmission line with random delays.



**Figure 8.2 Generic Testbench Architecture**

It is important **NOT** to underestimate the amount of verification required because it can be quite complex, and lengthy, with several thousands to millions of clock cycles. If verification makes use of BFM's, then those BFM's must reflect the behavior, and sometimes the timing, of the real interfaces.

**M** **Thumbs Up** Some useful guidelines in the design of a verifier or checker include the following:

- **Use a Validation Plan.** Because of the complexity of the tests, a test plan insures that the testbench includes all the necessary functions and validation tests. While this formalism might seem cumbersome to implement, it is generally the most efficient approach to insure, with a high level of confidence, that the tests will verify a design specification. A validation plan defines:

1. **WHAT** should be tested in the unit under test,
2. **METHODS** used for the tests,
3. **VALIDATION** approaches to ensure that the UUT meets the requirements.

The validation plan should define the classes of tests, the test mechanisms, and the validation approaches. The test plan does specify the actual test vectors. The test scenarios define the lists of tests to be performed for verification. In addition to testing normal operation, it is very important to test incorrect operations, such as illegal input combinations and sequences. The clearest method to define such tests is to build a table defining the requirements and the tests required to verify such requirements. The error injector model described in section 6.3 could be used to inject errors into the UUT.

- **List Errors to be Detected**

This list of errors to be detected provides the following advantages:

1. It defines the testbench validation results.
2. It guides UUT designers for errors to detect.
3. It enhances the testbench design review process to ensure completeness of tests.

- **Use Pseudo-Random Transactions**

It is important to emulate the randomness of the transactions that is subjected to the UUT. Transactions do not imply the nanosecond delays, but rather the arrival of different demands of the UUT at the different interfaces (e.g., DMA requests, Memory reads, memory writes, and idles). Pseudo-random events and relationships between events emulate the real environment of the UUT, where transactions occur in a random manner. The random functions should be portable among simulators, and should avoid the use of the *real* type. The pseudo-random transactions concept is demonstrated in section 8.6.

- **Use of Pseudo-Random Data**

To verify against any data dependencies in the design, it is also important to emulate randomness in the UUT data patterns. Common techniques to generate such random data are the use of random functions or Linear Feedback Shift Registers (LFSR). Random functions are practical for objects of integer types. However, LFSR are more practical for objects of *Bit\_Vector*, *Std\_Logic\_Vector*, *Unsigned* or *Signed* types of arbitrary widths (e.g., 2 to 300 bits). Section 5.4 describes the LFSR package available on disk. Section 5.5 describes the random package. Section 8.6 provides an example that makes use of the random package for the generation of random transactions, and the LFSR package for the generation of random data.

- **Use Timing Checkers**

If timing verification is an integral part of the tests, then timing checkers can be used to check for timing violations. The *Vital\_Timing* package can be used to provide the timing checks on the interfaces of the models. The *Vital\_Timing* package will be accelerated if the design is entirely an interconnection of primitives modeled according to the VITAL specifications. However, it is still recommended to use this package to verify the timing of a design (at any modeling level) for modeling style consistency, reuse, and potential acceleration.

Note that some users rely on static timing analyzer tools, instead of simulation, to verify the UUT's timing margins.

- **Use Procedures and Functions**

Procedures and functions tend to shorten the code for the verification. For example, for a very complex protocol, one could use several sets of procedures for each layer of the protocol, with each procedure driving the stimuli signals. In addition, one could create a high-level model of the protocol that is useful as a reference (see section 8.5).

- **Use Independent Engineers**

These verification tests are best implemented by engineers who are not directly involved in the design of the UUT. This approach avoids misunderstandings of the specifications.

### 8.3 REGRESSION TESTS

**Q** | What techniques can be used for regression tests?

**A** | Regression tests are tests performed on a design initially verified, but modified as a result of synthesis, design optimization, post-layout process, testability insertion, etc. What is important is assurance that the translated designs have maintained the original, tested functionality. Ideally, it would be desired to speed-up the regression tests, since extensive time is spent to verify the original design. Several approaches are used for these regression tests.

#### 8.3.1 File Compare Method ☺☺

The traditional approach is to create a golden stimulus and response file by storing the simulation results of the UUT interface signals. Any subsequent regression test creates a new set of stimulus and response file, also known as *LST* files. A file compare program, such as *Unix diff*, can be used to compare the results. This approach is NOT recommended because:

1. The simulation is slowed down because of TextIO process (see Section 9.1).
2. It is difficult to resolve the hardware errors locations when a mismatch occurs between the golden file and the new file. This is because the large amount of data collected makes analysis difficult without the help of custom designed software.
3. Files can be very large, and may exceed the available disk storage. A design typically consumes a very large amount of storage because of the various views of the design (synthesis outputs, routes, layout, schematics, etc.). Experience has shown that it is not uncommon to exceed the allocated disk space.

### 8.3.2 Design Verifier Method ☺☺

This approach makes use of a verifier (or checker) that monitors the performance of the design, and alerts the user to errors. Those errors are either displayed on the screen, and/or are stored in a file. In many instances, errors are injected into the stimulus vectors to verify proper error detection by the design. The verifier detects these errors as expected errors because it has knowledge of the tests and the expected responses. This data reduction technique provides several advantages over the file compare method:

1. The simulation is accelerated because TextIO is not heavily used.
2. Any design error can quickly be located because the verifier alerts the user of the class of error, the time the error occurred, and debug information about the error. If the severity of the error is high, the verifier could use the *assert* statement with a severity level of *failure* to stop the simulator when the fault occurs.
3. The log files are small and manageable because they record real errors.

### 8.3.3 MISR Method ☺☺

A MISR is a Multiple Input Signature Register. A MISR is placed external to the UUT in the testbench. All the interfaces of the UUT are connected to the MISR. A simulation is run with the golden version of the model, and a signature is recorded at the end of the simulation period. During subsequent regression tests, the newly generated signature is compared to the golden signature from within the testbench, therefore eliminating any TEXT IO or file transaction.

The auto-regression test approach with a single signature comparison at the end of a simulation run is very efficient because file IOs are not used. It is suitable when delivering models to a user who needs to only verify that the functionality of a model has not changed. However, there may be two concerns with this approach:

1. The potential possibility (even though, very remote) that two functional different designs yield the same signature.
2. Failure to identify approximately when a MISR comparison mismatch occurred.

As a compromise between speed and fault isolation (time after which a failure occurred), a set of signatures can periodically be stored in a file every  $n$  clocks, when the value of  $n$  is user defined. With every regression test, a new set of signatures is generated. A comparison between the set of signatures generated by the golden model and the set generated by the updated model can be made to detect any functional dissimilarities. For example, in a simulation of one million clocks, the writing of a MISR signature every 1000 clocks does not significantly impose a heavy speed penalty, but provides a debug simulation time zone to within 0.1% (1000 clocks) of the whole simulation time. The multiplicity of signatures significantly reduces the chance that two different functional designs yield the same signatures.

Data is strobed into the MISR when the data is stable, typically at a clock edge. The MISR recognizes all the states of type *Std\_Logic*. The MISR is based on the same principle as Built-In Self Test (BIST), but is used for the simulation rather than on the chip. Section 5.3 describes the MISR Package provided by Sandi Habinc from the European Space Agency.

### 8.3.4 Formal Verification Method

Formal verification tools such as the Synopsys® *Design Compare* or Compass *Vformal* can be used to verify that two circuits (the verified and the modified versions) are functionally equivalent. For large designs, the comparisons may need to be performed on a subsection by subsection basis because of tool limitations. It is also necessary to combine this formal verification technique with static timing analysis to verify that the timing constraints are still met.

## 8.4 FORMAL VERIFICATION

**Q** | What is formal verification? What are the advantages of formal verification? How does it work? How does it fit in the design cycle?

**A** | The following discussion was provided by *Compass Design Automation*, a company that offers a tool called *Vformal*®<sup>15</sup>. Formal verification is a solution for an HDL (e.g., VHDL) that can be used for two different purposes:

1. Specification checking to debug an original specification, or
2. Equivalence checking to compare two designs functionally.

### 8.4.1 How Formal Verification Works

Formal verification tools use rigorous mathematical methods to prove either that two VHDL designs are equivalent for all possible sets of inputs (equivalence checking), or that the original specification contains certain important features (specification checking).

When used for equivalence checking, the verification tool analyzes two designs in order to extract two representations of logical functionality in terms of Boolean equations. These representations can be thought of as logic trees. Next, the two logic trees are exhaustively compared to prove equivalence. If the two designs being compared are not equivalent, the tool identifies exactly how they differ.

When used for specification checking, the verification tool first analyzes the specification to extract its Boolean function, then it reads the properties that are to be checked. Properties are generally of two types - safety and liveness. Safety properties are events, or sequences of events, that the designer wants to ensure do not happen. Liveness properties are events, or sequences of events, that the designer desires to happen. The tool then uses logic tree traversal techniques to check that the properties are met in the specification. If the properties are not met, the tool generates a counter example to show how the specification failed.

### 8.4.2 How Formal Verification Fits in the Design Cycle

For **specification checking**, formal verification can be used to debug a specification early in the design cycle, even before synthesis, where changes are far less costly to make. For instance, specification checking can be used to locate and find such hard to locate problems as:

- Bus contention
- Protocol failure
- Unreachable states
- State machine lock-up
- Asynchronous feedback (often causing oscillating loops)
- Non toggling and single transition bits
- State bits and memory elements that do not affect the output
- Redundant state bits and memory elements
- Unnecessary I/O
- Incomplete sensitivity lists
- Range violations

For **equivalence checking**, formal verification can be used to check any design changes that naturally occur throughout the design cycle that have been traditionally verified with simulation. These changes could be due to:

- Engineering Change Orders (ECO)
- Iterative refinements
- Partitioning
- Re-synthesis
- Retargeting to a different technology (e.g., FPGA to ASIC)
- Optimization
- Test insertion
- Design reuse
- Reverse engineering
- Last-minute tweaking

### 8.4.3 Formal Verification Advantages

Among the major benefits of formal verification is the ability to analyze a design's logical structure to identify exactly where logic design errors exist. Simulation, in contrast, cannot automatically locate design errors. Using simulation as a debug methodology requires four steps of human intervention and intuitive guessing: vector creation, net probing, trace analysis, and problem diagnosis. Because of this guesswork, simulation may need to be repeated many times to isolate and fix design errors. Using formal verification, errors may be directly identified, thus reducing the number of iterations required for design verification. Formal verification tools can also prove the absence of errors, which simulation cannot do.

Using simulation, it is necessary for the designer to imagine a set of test vectors that will thoroughly test the design. Complete coverage is nearly always impossible because test vector size increases exponentially with the number of inputs and state bits. Thus, for most reasonable size designs, the time required to create the test vectors and run the simulation is prohibitive. Formal verification can achieve complete coverage without test vectors, and therefore provides results far faster than comprehensive simulation. Therefore, formal verification should be seriously considered for any complete verification methodology.

## 8.5 BUS FUNCTIONAL MODEL (BFM) MODELING

**Q** | What modeling styles are recommended for the modeling of BFMs?

**A** | Functional models (FM) and BFMs are very useful because they enable the modeling of the interfaces of a complex components, thus providing a "simulatable specification" of the interfaces. Unlike FMs, BFMs do not necessarily emulate the operations of the component, and therefore can be built relatively quickly, and can be quite efficient from a simulation viewpoint. FMs and BFMs are very useful in testbenches where they can be used to verify the operation of the units under test (UUT) when exposed to the interface signals. BFMs are generally easier to construct because the internal complexity of highly integrated devices are generally known by the vendor only. However, the signal interfaces and timing parameters are defined in vendor's specification sheets (even though they may not necessarily be sufficient enough to create a BFM). The functional model (BFM, and sometimes the FM) operational requirements include the following:

1. It must have a means to issue transactions (i.e., the scenarios) on the interface signals (e.g., READ transactions, WRITE transactions, IDLE transactions).
2. It must reflect the correct cycle timing (i.e., phase and clock cycle delays) on the interfaces.
3. It should reflect the correct propagation delays on the interface signals. These delays should be selectable for minimum, typical, maximum, or user defined delays.
4. It must react to the environment just like a real device. For example, if a user wishes to issue 1000 READ and 1000 WRITES transactions, and a Direct Memory Access (DMA) request signal is asserted by the environment after the first transaction, then the BFM must acknowledge the DMA, tri-state the appropriate signals, and stop issuing bus transactions until bus ownership is returned to the microprocessor.
5. If data flow from one model to another model is a requirement, then a means must be provided to transfer the data across the BFMs.
6. It should be able to force bus errors to verify the operations of the UUTs when exposed to errors (e.g., parity errors).

7. It should detect and report bus transaction errors. Note that some of the verification and error reporting functions may be performed by concurrent VHDL statements (e.g., components, procedures), and can be used collectively with the FM/BFM bus error monitors to implement the full error detection requirements of the model.
8. It should be able to log the bus transactions into a file, in a human readable format, for further analysis of the tests run on the UUT.
9. It may emulate the execution of instructions if mandated by the project.

Two generally accepted modeling approaches for the control of the BFM (and sometimes the FM) scenario generator are used (see Figure 8.5-1):

1. **Instruction file command format** where the instructions and modeling environment are defined in text files. The BFM model reads the files, interprets the instructions, and executes, in VHDL, the bus protocols necessary to achieve these instructions.
2. **Architectural command format** with in-line VHDL code. This approach may use files as source of data or high level instructions. In its simplest practical form, it includes concurrent VHDL statements to execute the desired bus transactions.

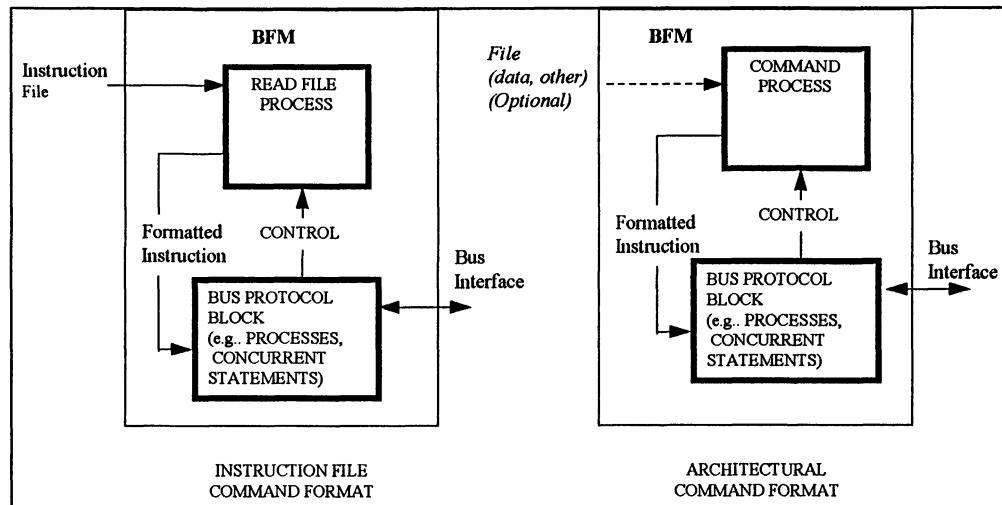
One possible architecture for the **instruction file command format** consists of two concurrent statements:

1. **Read File Process** reads instructions or data from a file (usually using Std.TextIO) under the throttle or handshaking control of a Bus Protocol Block. This process also formats the *TextIO* data into the proper format and data type for easy handling by the Bus Protocol Block.
2. **Bus Protocol Block** understands the bus protocol, instructs the Read File Process to read an instruction or data from a file, and accepts the formatted instructions from the Read File Process. The Bus Protocol Block then executes the commanded instruction using the interface protocol. The Bus Protocol Block may consist of concurrent statements including processes, concurrent signal assignments, and components.

The **architectural command format** style is very similar to the **instruction file command format**, with the exception that no file is read for the control of the transactions. Transaction or data information is sourced from within the command process in VHDL. Files may be used to transfer additional data or other information to the BFM. The **architectural command format** basically consists of two blocks:

1. **Command Process** executes VHDL code under the control of a Bus Protocol Block. This Command Process computes and formats the sequences of instructions to be handshaked with the Bus Protocol Block. These sequences are written in VHDL, thus taking full advantage of the features of the language. The Command Process may still access data or instructions from an external file if required. .

2. **Bus Protocol Block** understands the bus protocol and instructs the Command Process to supply more formatted information. The Bus Protocol Block decodes and executes the commanded instruction using the interface protocol.



**Figure 8.5-1 Two Potential Architectures for BFM Modeling**

There are variations to these architectures. For the **instruction file command format**, the Read File Process can be substituted by a procedure called by the Bus Protocol Process. However, this is **not recommended** because there is a lack of distinct separation between the bus protocol portion of the BFM and the actual instruction gathering process.

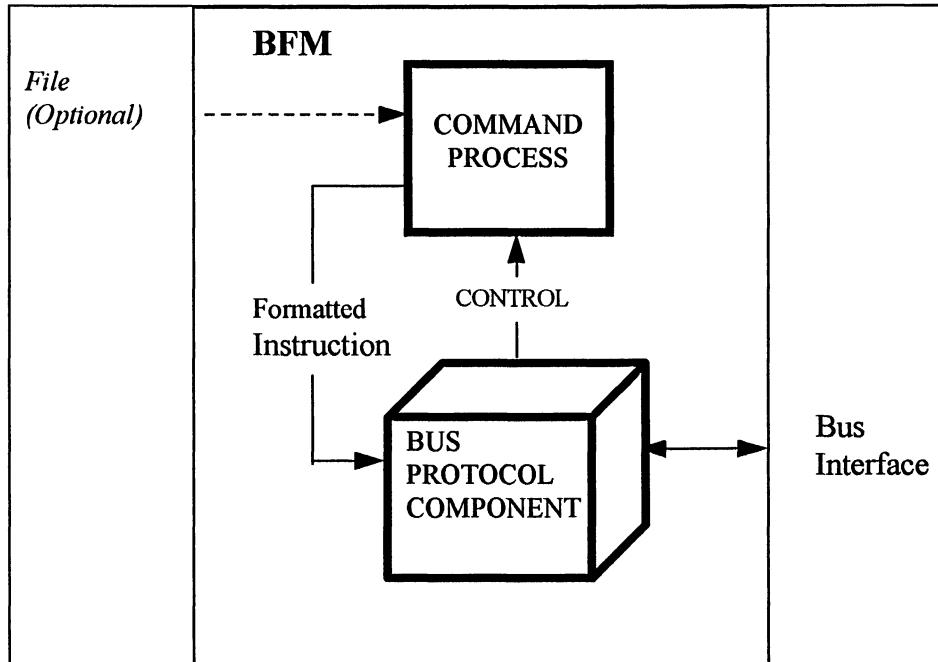
An interesting variation to the basic architecture for the **architectural command format** is the conversion of the Bus Protocol Block into a Bus Protocol Component as shown in Figure 8.5-2. This approach provides several advantages:

1. **Functional Separation.** The component clearly separates the **bus protocol** function from the **instruction or data** generation function. The architecture of the component has limited visibility on only the ports of the components. This is unlike concurrent statements that have visibility over all the ports of the BFM entity and all the signals of the BFM architecture.
2. **Flexibility and Adaptability to changes.** A change to the bus protocol, with no change to the interface ports, can easily be handled with a configuration declaration that maps the architecture of this component to a newly developed architecture.
3. **Information Hiding.** A component represents a level of hierarchy and represents a single instantiation into the BFM architecture. A process is represented with in-line code that can clutter the readability of the BFM.



It is recommended that the bus protocol portion of a BFM design be represented with a component instead of processes.

*Rationale:* See above mentioned advantages.



**Figure 8.5-2 Architectural Command Format with Bus Protocol Component**

An example of the architectural format with the bus protocol component is presented in the next section.

## 8.6 APPLICATION OF MISR, RANDOM, LFSR PACKAGES FOR AUTO-REGRESSION

**Q** | How can the MISR, random, and LFSR packages be applied for auto-regression testing?

**A** | As explained in section 8.3.3 And in section 5.3, the Multiple-Input Signature Register (MISR) package defines a concurrent procedure that implements a variable length MISR. The MISR essentially provides a signature of all the inputs transferred to the MISR. The MISR accepts inputs of type *Std\_Logic*; each *Std\_Logic* value has a unique four-bit vector associated with it. Thus, when the MISR is sampled, it will provide a signature that is dependent on the *Std\_Logic* typed values asserted on the MISR input signals. The length is determined by the length of input, ranging from 4 to 100. The input can be sampled on either Clock edge or both edges, delayed by clock

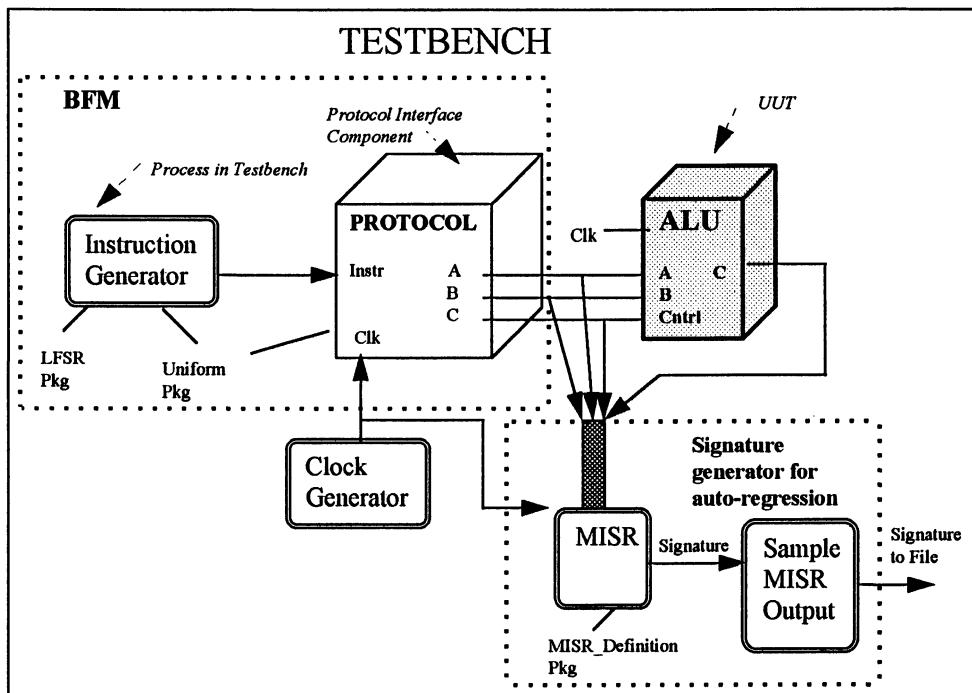
sense input, and selected with the rising and falling parameters. If neither option is selected, events on the input will determine the sampling point. For lengths greater than 100 bits, multiple MISRs can be used.

Figure 8.6-1 demonstrates the design of a testbench with the use of a MISR, and the generation of random data applied at random clock cycles, and with random delays. The testbench employs the methodology described in section 8.5 for the definition of a BFM. In this example, the UUT is an ALU. It accepts two eight-bit data inputs (*A* and *B*), three-bit of control (*Cntrl*), a clock (*Clk*), and produces an eight-bit output (*C*). Depending upon the values of the control signals, this ALU produces either arithmetic and logical operations, or tri-state outputs, or 'U's and 'X's. All outputs are registered.

To demonstrate the BFM design methodologies, a BFM is used to stimulate this UUT. This BFM consists of a PROTOCOL component and an instruction process. The PROTOCOL component accepts a high level instruction from the INSTRUCTION process as a record. It decomposes this instruction, and translates it into the low level protocol interface signals to the UUT. In this simple example, the instruction supplies the data to be asserted onto the ALU. The PROTOCOL component transfers the requested data onto its outputs for assertion to the UUT. The PROTOCOL component uses the random package to introduce nanosecond random delays in the application of those values.

The instruction generation process uses the LFSR package to supply random data to the PROTOCOL component. In addition, to demonstrate the concept of randomness in the application of the transactions, the random package is used to introduce random clock cycles delays in the definition of each of the test vectors.

The testbench makes use of the MISR concurrent procedure defined in the *MISR\_Definition* package. All the signals surrounding the ALU make up the MISR input. The MISR is clocked with the rising edge of the clock. The MISR output is sampled by a process every 200 clocks, and this sample is stored into a file for later comparison with a different architecture of ALU design. This testbench does not include a verifier.



**Figure 8.6-1 Testbench Architecture**

The compilation order for the various files is shown in figure 8.6-2.

FILE	DESCRIPTION
verifc\types_p.vhd	Type definitions for model
verifc\uniform.vhd	Uniform random procedure in a package
package\image_pb.vhd	Image package
verifc\lfsrstd.vhd	LFSR package
package\miser_pb.vhd	MISR package
ieee\numstd.vhd	Numeric package
verifc\alu_ea.vhd	ALU model
verifc\protocol.vhd	Protocol Model
verifc\alu_tb.vhd	ALU testbench

**Figure 8.6-2. Compilation Order and Files Used in Testbench Design**

Figure 8.6-3 represents a package for the definition of a record type. Figure 8.6-4 represents a package for an *integer* based (rather than *real*) definition of a random number generator for enhanced portability.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

package Types_pkg is
  type Instr_Typ is record
    A          : Std_Logic_Vector(7 downto 0);
    B          : Std_Logic_Vector(7 downto 0);
    Cntrl     : Std_Logic_Vector(2 downto 0);
    Reset     : Boolean;
  end record;
end Types_pkg;

```

**Figure 8.6-3 Package for Definition of a Record Type.(verifc\types\_p.vhd)**

```

package Uniform_Pkg is
  procedure Random(variable Seed_v      : inout Integer;
                    constant Modulus_c : in   Integer);
end Uniform_Pkg;

package body Uniform_Pkg is
  procedure Random(variable Seed_v      : inout Integer;
                    constant Modulus_c : in   Integer) is
    constant Multiplier_c : Integer := 25173;
    constant Increment_c  : Integer := 13849;
  begin
    Seed_v := (Multiplier_c * Seed_v + Increment_c) mod
               Modulus_c;
  end Random;
end Uniform_Pkg;

```

**Figure 8.6-4 Package for Definition of Integer Based Random Number Generator(verifc\uniform.vhd)**

Figure 8.6-5 is a model of the ALU.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.NUMERIC_STD.all;

entity Alu is
  port(A          : in  Std_Logic_Vector(7 downto 0);
        B          : in  Std_Logic_Vector(7 downto 0);
        Cntrl     : in  Std_Logic_Vector(2 downto 0);
        Clk        : in  Std_Logic;
        C          : out Std_Logic_Vector(7 downto 0));
end Alu;

```

```

architecture Alu_a of Alu is
  signal C_s : Unsigned(7 downto 0);
begin -- Alu_a
  Alu_Lbl : process (A, B, Cntrl)
    variable A_v : Unsigned(7 downto 0);
    variable B_v : Unsigned(7 downto 0);
  begin -- process Alu_Lbl
    A_v := Unsigned(A);
    B_v := Unsigned(B);

    case Cntrl is
      when "000" =>
        C_s <= A_v and B_v;
      when "001" =>
        C_s <= A_v or B_v;
      when "010" =>
        C_s <= (7 downto 5 => 'U',
                 others       => 'X');
      when "011" =>
        C_s <= "00000000";
      when "100" =>
        C_s <= "11111111";
      when "101" =>
        C_s <= A_v + B_v;
      when "110" =>
        C_s <= A_v - B_v;
      when "111" =>
        C_s <= (others => 'Z');
      when others =>
        C_s <= (3 downto 0 => 'U',
                 others       => 'X');
    end case; -- Cntrl
  end process Alu_Lbl;

  Reg_Lbl : process
  begin -- process Reg_Lbl
    wait until Clk'event and Clk = '1';
    C <= Std_Logic_Vector(C_s);
  end process Reg_Lbl;
end Alu_a;

```

**Figure 8.6-5 ALU Model (Verifc\alu\_ea.vhd)**

Figure 8.6-6 is a model of the protocol component. This protocol component is simple in operation. However, it is used as a model to demonstrate the concepts of using a protocol component commanded by an instruction. This instruction can be issued by either a command process or command component. This component reads the instruction, and if RESET is requested, then the outputs are reset to zeros. Otherwise, the outputs are those defined in the instruction record. This component also adds random delays in the output signals.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
library Work;
use Work.Types_pkg.all;
use Work.Uniform_Pkg.all;

entity Protocol is
  port(Instr : in  Work.Types_Pkg.Instr_Typ;
        Clk   : in  Std_Logic;
        A     : out Std_Logic_Vector(7 downto 0);
        B     : out Std_Logic_Vector(7 downto 0);
        Cntrl : out Std_Logic_Vector(2 downto 0));
end Protocol;

architecture Protocol_a of Protocol is
  constant Mdls_c : Natural := 35; -- time delay modulus
begin
  Data_Lbl: process
    variable DlyA_v : Natural := 5;
    variable DlyB_v : Natural := 7;
    variable DlyC_v : Natural := 13;
  begin
    wait until Clk'event and Clk = '1';
    Work.Uniform_Pkg.Random(Seed_v      => DlyA_v,
                            Modulus_c  => Mdls_c);

    Work.Uniform_Pkg.Random(Seed_v      => DlyB_v,
                            Modulus_c  => Mdls_c);

    Work.Uniform_Pkg.Random(Seed_v      => DlyC_v,
                            Modulus_c  => Mdls_c);

    if Instr.Reset then
      A    <= (7 downto 5 => '1',
              others      => '0') after DlyA_v * 1 ns;
      B    <= (2 downto 0 => '1',
              others      => '0') after DlyB_v * 1 ns;
      Cntrl <= (others  => '0') after DlyC_v * 1 ns;
    else
      A    <= Instr.A after DlyA_v * 1 ns;
      B    <= Instr.B after DlyB_v * 1 ns;
      Cntrl <= Instr.Cntrl after DlyC_v * 1 ns;
    end if;
  end process Data_Lbl;
end Protocol_a;

```

**Figure 8.6-6 Protocol Component (verifc\protocol.vhd)**

Figure 8.6-7 represents the testbench that incorporates the UUT, the protocol component, and the instruction generator.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

library Work;
use Work.MISR_Definition.all;
use Work.LfsrStd_Pkg.all;
use Work.Uniform_Pkg.all;
use Work.Types_pkg.all;

entity ALU_TB is
begin
end ALU_TB;

architecture ALU_TB_A of ALU_TB is
component Alu
port(
    A      : in      Std_Logic_Vector(7 downto 0);
    B      : in      Std_Logic_Vector(7 downto 0);
    Cntrl  : in      Std_Logic_Vector(2 downto 0);
    Clk    : in      Std_Logic;
    C      : out     Std_Logic_Vector(7 downto 0)
);
end component;

component Protocol
port(
    Instr      : in      Work.Types_Pkg.Instr_Typ;
    Clk        : in      Std_Logic;
    A          : out     Std_Logic_Vector(7 downto 0);
    B          : out     Std_Logic_Vector(7 downto 0);
    Cntrl      : out     Std_Logic_Vector(2 downto 0)
);
end component;

constant Delay_c : Natural := 15;
signal A      : Std_Logic_Vector(7 downto 0) := "10100000";
signal B      : Std_Logic_Vector(7 downto 0) := "01010101";
signal Cntrl  : Std_Logic_Vector(2 downto 0) := (others => '0');
signal Clk    : Std_Logic := '0';
signal C      : Std_Logic_Vector(7 downto 0) := (others => '0');
signal Reset  : Boolean;
signal MISR_s : Std_Logic_Vector(26 downto 0) := (others => '0');
signal MISRin_s : Std_Logic_Vector(26 downto 0) := (others => '0');
signal Instr   : Work.Types_Pkg.Instr_Typ
                := (Reset => True,
                    A      => (others => '0'),
                    B      => (others => '1'),
                    Cntrl => (others => '0'));

begin
    -- UUT component
    Alu_i: Alu
    port map (
        A      => A,
        B      => B,
        Cntrl  => Cntrl,
        Clk    => Clk,
        C      => C
    );

```

```
-- Protocol component. Receives instructions from command processes.
-- Drives the UUT
Protocol_i: Protocol
  port map (
    Instr          => Instr,
    Clk            => Clk,
    A              => A,
    B              => B,
    Cntrl          => Cntrl
  );

-- Command concurrent statement to instruct RESET.
Instr.Reset  <= True,
             False after 300 ns;

-- Command process to instruct A data. Note random delay
-- in the assertion of the command
DataA_Lbl: process
  variable Lp_v : Natural := 3;
begin
  Work.Uniform_Pkg.Random(Seed_v      => Lp_v,
                          Modulus_c   => Delay_c);
  Clk_Lbl : for I in 0 to Lp_v loop
    wait until Clk'event and Clk = '1';
  end loop Clk_Lbl;

  Instr.A  <= Work.LfsrStd_Pkg.LFSR(A);
end process DataA_Lbl;

-- Command process to instruct B data. Note random delay
-- in the assertion of the command
DataB_Lbl: process
  variable Lp_v : Natural := 2;
begin
  Work.Uniform_Pkg.Random(Seed_v      => Lp_v,
                          Modulus_c   => Delay_c);
  Clk_Lbl : for I in 0 to Lp_v loop
    wait until Clk'event and Clk = '1';
  end loop Clk_Lbl;

  Instr.B  <= Work.LfsrStd_Pkg.LFSR(B);
end process DataB_Lbl;

-- Command process to instruct Cntrl data. Note random delay
-- in the assertion of the command
DataC_Lbl: process
  variable Lp_v : Natural := 1;
begin
  Work.Uniform_Pkg.Random(Seed_v      => Lp_v,
                          Modulus_c   => Delay_c);
  Clk_Lbl : for I in 0 to Lp_v loop
    wait until Clk'event and Clk = '1';
  end loop Clk_Lbl;

  Instr.Cntrl  <= Work.LfsrStd_Pkg.LFSR(Cntrl);
end process DataC_Lbl;
```

```

-- Concurrent statement to compute input to MISR
-- All signals of interest are used in the MISR
MISRin_s <= C & A & B & Cntrl;

-- MISR computation -- Concurrent procedure
Work.MISR_Definition.MISR(
    Clk          => Clk,      -- Sample clock
    Reset        => Reset,     -- Boolean; Reset of MISR
    Input         => MISRin_s, -- MISR input
    MISR         => MISR_s); -- inout MISR
    -- constant Rising:   in Boolean := True;
    -- constant Falling: in Boolean := False;
    -- constant HeaderMsg: in String  := "MISR:";
    -- constant Sense:    in Time    := 0 ns);
    -- Sense time after clock

-- Write MISR data to file every 200 clock cycles
WriteMisr_Lbl : process
use Work.Std_Language_TextIO.all;
use Std.TextIO.all;
file MISR_f : Std.TextIO.Text is out "misr_out.txt";
variable Line_v : Std.TextIO.Line;
variable Count_v : Natural := 0;

begin -- process WriteMisr
wait until Clk'event and Clk = '1';
if Count_v /= 200 then
    Count_v := Count_v + 1;
else
    Count_v := 1;
    Std.TextIO.Write(Line_v, now);
    Std.TextIO.write(Line_v, string'("  "));
    Work.Std_Language_TextIO.Write(Line_v, MISR_s);
    Std.TextIO.WriteLine(MISR_f, Line_v);
end if;
end process WriteMisr_Lbl;

-- Clock generation
Clk  <= not Clk after 50 ns;

-- MISR Reset
Reset <= True,
        False after 500 ns;
end ALU_TB_A;

```

**Figure 8.6-7 Testbench (verifc\alu\_tb.vhd)**

Figure 8.6-8 represents some simulation listing of the testbench simulation.

ns	delta	a	b	cntrl	clk	c	reset	mistr_s	instr
0	+2	10100000	01010101	000	0	00000000	true	11111111111111111111111111111111	(00000000, 11111111, 000, true)
50	+2	10100000	01010101	000	1	00000000	true	001111010011110000111010	(00000000, 11111111, 000, true)
60	+1	10100000	000000111	000	1	00000000	true	001111010011110000111010	(00000000, 11111111, 000, true)
79	+1	11100000	000000111	000	1	00000000	true	00111010011110000111010	(00000000, 11111111, 000, true)
100	+0	11100000	000000111	000	0	00000000	true	00111010011110000111010	(00000000, 11111111, 000, true)
150	+2	11100000	000000111	000	1	00000000	true	010010100001001111000000	(00000000, 11111111, 000, true)
200	+0	11100000	000000111	000	0	00000000	true	010010100001001111000000	(00000000, 11111111, 000, true)
250	+2	11100000	000000111	000	1	00000000	true	101000110110000100111111	(00000000, 11111111, 000, true)
300	+0	11100000	000000111	000	0	00000000	true	101000110110000100111111	(00000000, 11111111, 000, false)
350	+2	11100000	000000111	000	1	00000000	true	001101001111011000010000	(00000000, 11111111, 000, false)
360	+1	00000000	000000111	000	1	00000000	true	001101001111011000010000	(00000000, 11111111, 000, false)
362	+1	00000000	11111111	000	1	00000000	true	001101001111011000010000	(00000000, 11111111, 000, false)
400	+0	00000000	11111111	000	0	00000000	true	001101001111011000010000	(00000000, 11111111, 000, false)
450	+2	00000000	11111111	000	1	00000000	true	01101000011110000110100	(00000000, 11111111, 000, false)
500	+0	00000000	11111111	000	0	00000000	false	01101000011110000110100	(00000000, 11111111, 000, false)
...									
4250	+2	10000000	01010111	011	1	00000000	false	000010100000110100101011	(11000000, 00101011, 011, false)
4272	+1	11000000	01010111	011	1	00000000	false	000010100000110100101011	(11000000, 00101011, 011, false)
4281	+1	11000000	00101011	011	1	00000000	false	000010100000110100101011	(11000000, 00101011, 011, false)
4300	+0	11000000	00101011	011	0	00000000	false	000010100000110100101011	(11000000, 00101011, 011, false)
4350	+2	11000000	00101011	011	1	00000000	false	100100101010110011100001	(11000000, 00010101, 011, false)
4400	+0	11000000	00101011	011	0	00000000	false	100100101010110011100001	(11000000, 00010101, 011, false)

### **Figure 8.6-8 Simulation List File**

Figure 8.6-9 provides the contents of the output file *mistr\_out.txt*.

20050	ns	000001001110010110001011010
40050	ns	110001000000001111101001110
60050	ns	110011000011000110001100001
80050	ns	011101011001001111010000100
100050	ns	000111011000001100001001000
120050	ns	000010011010010010001010100
140050	ns	101111000011111001001100110
160050	ns	100010011100100111001100000
180050	ns	110110111011100100001111100
200050	ns	101001111010110001101001110
220050	ns	1001001011110111101000100110
240050	ns	100000101111110010100100111

**Figure 8.6-9 Contents of Simulation Output File (verifc\misr\_out.txt)**

## 8.7 STRENGTH STRIPPER

- Q** Given a model written with *IEEE.Std\_Logic\_1164* strong signals for control (e.g., '0', '1'), how can this model be tested in a testbench that may assert weak signals (e.g. 'H', 'L') to its interfaces? Modifying the model is not a feasible consideration.
- A** The simplest solution is to use *IEEE.Std\_Logic\_1164* strength stripper function *To\_X01*. This type conversion function is put on the driving signals of the component port. Table 8.7 represents the four possibilities.

**Table 8.7 Port Mapping for Strength Stripper**

Port Mode	PORT MAPPING EXAMPLE	
in	<code>port map (In_Port =&gt; To_X01Z(NET_RESET));</code>	
out	<code>port map (To_X01Z(Out_Port) =&gt; NET_BUFFER_PORT);</code>	
buffer	<code>port map (To_X01Z(Buffer_Port) =&gt; NET_BUFFER_PORT);</code>	
inout	<code>port map (To_X01Z(InOut_Port) =&gt; To_X01Z(NET_INOUT_PORT));</code>	

Required to force outputs to 'X01Z' values

Required to force inputs into model to 'X01Z' values

## **9. POTPOURRI**

---

---

This section addresses a variety of topics including:

- Methods to enhance simulation speed
- Methods to access signals internal to a component
- Methods to transfer a line onto a signal
- Errors in type declaration in multiple packages
- Information available on the Internet
- Choice of editors
- Definition of Vital.
- Definition of behavioral modeling
- Sample VHDL final exam with solutions

## 9.1 METHODS TO ENHANCE SIMULATION SPEED

**Q** | How can VHDL simulations be accelerated?

**A** | Simulation speed<sup>16</sup> can be accelerated by minimizing the amount of work and calculations that the simulator must execute. This includes code that handles signals, concurrent statements, type conversions, subprograms, resolution functions, and run-time checks. It is important to reduce the memory requirements for the models to avoid disk swapping. It is also important to avoid memory leaks. Many of the ideas presented in this discussion were contributed by Sandi Habinc<sup>[2]</sup>, but represent the author's interpretation of those concepts.

The following guidelines generally apply to non-synthesizable models. The rules for writing code for synthesis are very strict, and are established for the sole purpose of optimizing the synthesis process. The synthesis rules do not necessarily map into the guidelines to achieve simulation speed-up.

### 9.1.1 Signals

Every object (variables and signals) carries with it information related to its characteristics, such as its base, length (for array), direction (for array), and range (for array). However, signals carry additional information required for simulation. This information varies among simulators, and is a function of how it processes the data. This additional information translates to increased simulation time because it must be processed by the simulator. Note that composite types (arrays, records) generate one signal for each element in the type. Thus, a 32-bit bus of type *Std\_Logic\_Vector* is equivalent to 32 individual signals. The information specific to signals includes the following:

- **Projected waveforms**, with calculations dependent upon the kind of delay defined in the signal assignment (i.e., *inertial*, *transport*, or *reject* delay).
- **Event list**, for computation of *Signal\_Name'event* and *Signal\_Name'last\_event*.
- **Activity list**, for computation of *Signal\_Name'active* and *Signal\_Name'last\_active*.
- **Signal list**, for computation of *Signal\_Name'last\_value*.
- **Drivers**, for the identity and value of each driver sourced by concurrent statements.
- **Resolution function**, for the computation of resolved values.

Every signal assignment, regardless of the value assigned, causes the simulator to compute and update some or all of this signal information. Therefore, to enhance simulation speed, it is important to minimize the number of signal assignments. To achieve this goal, the following guidelines are recommended.

### 9.1.1.1 Use Signals for Inter-Process Communication Only

**忌諱** Signals should be used for inter-process (or inter-concurrent statement) communication. Signals should not be used for inner-process communication. Thus, if a process needs to transfer data for its internal operations, it is more efficient to use variables instead of signals.

Avoid representing elements with large data structures (such as memories) as signals. The signals significantly slow down the simulator, and require two orders of magnitude more host memory than variables holding the same objects. Instead, use signals for the memory interface signals, and a variable for the storage of the memory array (see section 6.1.1).

### 9.1.1.2 Avoid Signal Reassignment for Same Value

**忌諱** Avoid reassignment of signals to the same value, as shown in figure 9.1.1.2. This concept is generally true; however, for some simple processes, simulator vendors may do other optimizations that overshadow this issue. For instance, process *Avoid\_Lbl* is so simple that Model Technology would essentially optimize it, whereas process *Do\_Lbl* contains a variable declaration that prevents the process from being optimized. However, for processes that are more complex, this approach would be more simulation efficient.

```
Avoid_Lbl: process      -- OK for synthesis
begin
  wait until Clk'event and Clk = '1';    --
  S <= Some_Signal;
end process Avoid_Lbl;

Do_Lbl: process
  variable SomeVar : Some_Type;
begin
  wait until Clk'event and Clk = '1';    --
  if Some_Signal /= SomeVar then
    S <= Some_Signal;
    SomeVar := Some_Signal;               ←
  end if;
end process Do_Lbl;
```

Figure 9.1.1.2 Signal Reassignment

### 9.1.1.3 Avoid Individual Assignment of Individual Elements of a Composite

**忌諱** In a process, avoid the signal assignment of each individual element of a record. Instead, copy the signal onto a variable, update the individual subelements of the variable (through variable assignments), and then make a signal assignment of the whole composite object. This is demonstrated in figure 9.1.1.3.

```

Test_Lbl: process(R_s)
    variable : R_v : Some_record_Typ; --
begin
    R_v := R_s;
    R_v.Element1 := SomeValue1;
    R_v.Element2 := SomeValue2;
    ...
    R_v.ElementK := SomeValueK;
    R_s <= R_v;   --
end process Test_Lbl;

```



In synthesis, either records are not allowed, or allowed with the fields limited to *Bit*, *Bit\_Vector*, *Std*, *Std\_Logic\_Vector*, *Boolean*, or *Integer*.

**Figure 9.1.1.3 Signal Assignment of Whole Object rather than its Subelements**

#### 9.1.1.4 Reduce Number of Signals

**All the time** Reduce the number of signals in the design and avoid unnecessary signals. Implicit signals ARE SIGNALS of predefined types and it is preferred to avoid them. Figure 9.1.1.4 demonstrates the concept of minimizing implicit signals, if it is avoidable.

```

Poor_Lbl: process(S)
begin
    if (S'delayed = '0' and S = '1') then
        assert False
        report "Rising edge of Signal S"
        severity Note;
    end if;
end process Poor_Lbl;

Better_Lbl: process(S )
begin
    if (S'last_value = '0' and S = '1') then
        assert False
        report "Rising edge of Signal S"
        severity Note;
    end if;
end process Better_Lbl;

```

Equivalent to S'delayed(0 ns).

S'delayed is an implicit signal

S'last\_value is a function , and  
not a signal

**Figure 9.1.1.4 Minimizing Implicit Signals**

#### 9.1.1.5 Avoid resolved signals

**All the time** Avoid resolved signals since they require a significant amount of overhead. The only exception to this rule is type *Std\_logic* and *Std\_Logic\_Vector* in package *IEEE.Std\_Logic\_1164* because these types are typically optimized on most simulators.

Avoid large vectors and/or complex records because VHDL resolves all signals down to the scalar level and requires a driver for each scalar element of the composite signal. Large complex records structures are often used for signals in performance level modeling. It would be desirable to have the next version of VHDL include variant records (a la Ada style) and atomic signals. This would drastically increase simulation performance when

the model is operating at the performance level.

### 9.1.2 Concurrent Statements

Each concurrent statement represents a primitive behavior element, and each invocation is costly in terms of processing. The following guidelines are recommended:

#### 9.1.2.1 Number of Processes/Concurrent Statements

**Rule** Keep the number of processes and concurrent statements to a minimum in order to reduce the simulation overhead. This can be achieved by grouping common functions within processes or concurrent statements. For example, all registers can be updated in a single process. Operations, sensitive to the same signals, can be grouped in the same concurrent statement, reducing the number of concurrent statements to invoke for each signal event. Figure 9.1.2.1 demonstrates this concept.

```
architecture Proc_a of Proc is
    signal S1, S2, S3, S4, S5a, S5b : Bit;
begin
    -- Bad use of concurrent statement ♫
    S1 <= not S2;      -- assuming that S1 and S3 are used as
    S3 <= S1 or S4;   -- temporary, intermediate signals.
    S5a <= not S3;

    -- Good use of concurrent statement ♪♪
    S5b <= not ((not S2) or S4); -- Temporary signal S3 is not used
end Proc_a;
```

**Figure 9.1.2.1 Minimizing the Number of Concurrent Statement**

#### 9.1.2.2 Concurrent Signal Assignment, Sensitivity list or Wait Statements<sup>[6]</sup>

For best simulation performance, it is best to use either a concurrent signal assignment, or a process that could by hand be trivially converted into a concurrent signal assignment. The performance of a process with *wait* statements depends upon the compiler's ability to analyze the *wait* expressions and procedure calls within the process. *Wait* statements do not necessarily have to be slower than sensitivity lists. *Wait* statements are more general. Through analysis, the compiler attempts to reduce each process to an equivalence (in semantic meaning) sensitivity list. If it does not detect this equivalence, then it may not be able to do other optimizations that could naturally apply to the sensitivity list case. For example, in a sensitivity list, the compiler knows that it is always sensitive to the list of signals. It also knows that it only has one (implied) *wait*. This allows the compiler to avoid having to dynamically change the list of signal expressions to which it is sensitive. In the general *wait* case, the *wait* must enable a set of wake-up signal conditions and must then suspend itself. This is generally more expensive because the simulator is tuned to fast simulation times, rather than fast process suspension. When the signal awakens it, it must restore its state, and verify that the *wait* is satisfied. If so, then it must remove the sensitivity to the signal expression, and continue execution. For processes with sensitivity list, there is no need for dynamic changing of the signal expressions.

In the example shown in figure 9.1.2.2-1, the sensitivity list is compared with a process having multiple wait statements.

```

architecture Speed_a of Speed is
    signal S1 : Natural;
    signal S2 : Natural;
begin
    Slower_Lbl: process
    begin
        wait on S1, S2;
        -- ... -- stage 0 processing
        wait on S2 until S1 = 5;
        -- ... -- stage 1 processing
    end process Slower_Lbl;

    Better_Lbl: process(S1, S2) -- Better for speed ☺☺
        variable State_v : Natural; -- Defines stage history
        constant MaxStates_c : Natural := 5;
    begin
        if (S1'event or S2'event) and (State_v = 0) then
            -- ... -- stage 0 processing
        end if;
        if (S2'event and S1 = 5) and State_v = 1 then
            -- ... -- stage 1 processing
        end if;
        ...
        State_v := (State_v + 1) mod MaxStates_c;
    end process Better_Lbl;
end Speed_a;

```

**Figure 9.1.2.2-1 Sensitivity List Versus Wait Statement (potpouri\speed.vhd)**

☺☺ In summary, processes using a sensitivity list are almost always better than ones with *wait* statements. Also, *Model Technology* is able to greatly accelerate concurrent signal assignments that are scalar expressions of Boolean operators on *Std\_Logic* type containing less than five signals. Other vendors may do similar optimizations. So it is advantageous to combine simple concurrent signal assignments into more complex ones, up to about 5 signals, and to convert simple processes into such concurrent signal assignments. Since acceleration techniques are vendor dependent, *Model Technology* suggests the following rules to accelerate processes:

- ☺☺
- Use variables instead of signals whenever possible.
  - Use sensitivity list, or one *wait* statement of the form *wait until CLK = '1'*;
  - It is acceptable to use generics and expressions of generics for delay on the right hand side of an assignment.

Figures 9.1.2.2-2 and 9.1.2.2-3 represents two examples of acceleratable processes. The first one also uses a generic. The second one has a wait statement. In either case the right hand side can be an expression.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

entity Crystal_Clock is
  generic (
    Period : time := 10 ns;
    Delay : time := 82 ns
  );
  port (
    Clock: out Std_uLogic -- direction demonstrates
           ); -- true intent of the model
end Crystal_Clock;

architecture Crystal2 of Crystal_Clock is
  signal ClockDelay : Std_uLogic := '1';
  signal Clk : Std_uLogic := '0';
begin
  Clock <= Clk; -- concurrent signal assignment

  Clockdelay <= '0' after Delay; -- concurrent signal assignment

  Clk2_Lbl : process (Clk, ClockDelay)
  begin
    Clk <= not (Clk or clockDelay) after Period/2;
  end process Clk2_Lbl;
end crystal2;

```

SIMULATION RESULTS				
ns	delta	clock	clockdelay	clk
0	+1	0	1	0
82	+0	0	0	0
87	+1	1	0	1
92	+1	0	0	0
97	+1	1	0	1
102	+1	0	0	0
107	+1	1	0	1

**Figure 9.1.2.2-2 Acceleratable Process with Sensitivity List  
(potpourri\crystal.vhd)**

```

... -- empty entity
architecture WT of WaitTest is
  signal Clk : Std_Logic := '0';
  signal B   : Std_Logic := '0';
  signal C   : Std_Logic := '0';
begin
  Clk <= not Clk after 10 ns;

  B1_Lbl: process -- sensitivity list is Clk
  begin
    wait until Clk ='1'; --Single wait of the form wait until Clk='1'
    B <= C after 3 ns;
  end process B1_Lbl;
end WT;

```

**Figure 9.1.2.2-2 Acceleratable Process with *Wait* (potpourri\wait.vhd)**

### 9.1.2.3 Processes with Multiple Clocks

**AVOID** Use separate processes for each clock to prevent the evaluation of statements due to unrelated clock signals.

### 9.1.2.4 Disabling Non-Necessary Concurrent Statements

In a testbench, several concurrent statements are typically used for debug or model verifiers. Not all those concurrent statements need to be active during every simulation

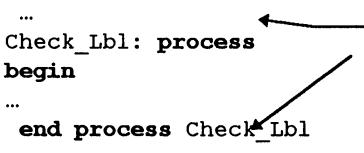
run. For example:

1. Verifiers are not used during initialization particularly when long scan chains are initialized, or if the model under test is in self-test.
2. Verifiers used for initial debug may not be needed for a regression test with MISR signature (see section 8.3)
3. TextIO logging of trace signals are not necessarily needed for certain tests.

 Those concurrent statements can be disabled or prevented from execution using the following alternative methods (ordered from most efficient to least efficient). Disabling the current statement with the *generate* statement is the fastest option to enhance simulation speed since the statement is never elaborated. However, if that is not acceptable, then waiting for a specified time duration (e.g., until end of initialization) prevents useless code from being executed. The *if* condition is similar in concept to the *wait* statement since it prevents the execution of a section of the code when the condition is false.

- *Generate statements* can exclude concurrent statements from participating in an elaborated design, thus eliminating all invocation costs. This is demonstrated below. 

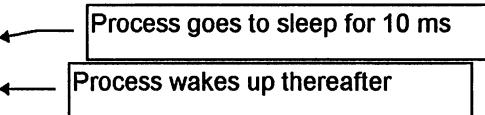
```
Checker_Lbl: if Enable_g generate
    ...
    Check_Lbl: process
        begin
            ...
        end process Check_Lbl
    end generate Checker_Lbl;
```



Any number of concurrent statements disabled if *Enable\_g* is False

- *Wait for* time can be used to wait for an initialization time period prior to enabling the required process as shown below.

```
Checker_Lbl: process
    constant InitTime_c : Time := 10 ms;
begin
    wait for InitTime_c;
    ForEver_Lbl: loop
        wait on S1, S2;
        ... -- Checker code
    end loop ForEver_Lbl;
end process Checker_Lbl;
```



- **Conditional statement** can protect a design from execution. The outer conditional statements should reduce the necessity to execute the enclosed statements if the condition is false. Branches based on the highest probability of occurring should be coded first. This method does not achieve the same goals as the other two methods, since the process is invoked whenever the sensitivity list is stimulated. This method is demonstrated below.

```

Checker_Lbl: process(S1, S2)
begin
    if CheckerOn_g then
        -- Code for checker is here
    end if;
    -- Other code inserted here
end process Checker_lbl

```

Process enabled by a generic of type Boolean. Process wakes up on every event on S1 or S2.

### 9.1.2.5 Avoid Repeated Code or Function Calls

 Some code includes redundant complex statements in multiple paths. Reduce the computations in those redundant paths by saving temporary results in variables. Figure 9.1.2.5-1 provides an example of such redundant code, and figure 9.1.2.5-2 demonstrates a method to improve efficiency.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity Redundancy is
    port (X      : in     Std_Logic_Vector(1 downto 0);
          B      : in     Std_Logic_Vector(1 downto 0);
          D      : in     Std_Logic_Vector(1 downto 0);
          E      : in     Std_Logic_Vector(1 downto 0);
          A      : out    Std_Logic_Vector(1 downto 0);
          C      : out    Std_Logic_Vector(1 downto 0);
          G      : inout  Std_Logic_Vector(1 downto 0));
end Redundancy;

architecture Redundancy_a of Redundancy is  -- Dummy code
function Some_Function(X : Std_Logic_Vector)
    return Std_Logic_Vector is
    variable R_v : Std_Logic_Vector(X'length - 1 downto 0);
begin
    R_v := (X(X'left) or X(X'right)) &
            (X(X'left) and X(X'right));
    return R_v;
end Some_Function;

function Some_Function2(X : Std_Logic_Vector)
    return Std_Logic_Vector is
    variable R_v : Std_Logic_Vector(X'length - 1 downto 0);
begin
    R_v := (X(X'left) and X(X'right)) &
            (X(X'left) or X(X'right));
    return R_v;
end Some_Function2;

```

```

begin
  Redundant_Lbl: process(X, B, D, G, E)
    begin
      if Some_Function(X) = "00" then
        A <= Some_Function2(B);
        C <= D;

      elsif Some_Function(X) = "01"
        and G = "11" then
        A <= Some_Function2(B);
        C <= E;

      else
        A <= Some_Function2(B);
        C <= "00";
      end if;

      G <= Some_Function2(B) or D;
    end process Redundant_Lbl;
end Redundancy_a;

```

1. *Some\_Function* may be used twice (efficiency issue)
2. *Some\_Function2* is used twice (efficiency issue)
3. *A* is assigned same value in three places (readability issue)



**Figure 9.1.2.5-1 Example of Redundant Code (potpouri\redund.vhd)**

```

Better_Lbl: process(X, B, D, G, E)
  variable X_v : Std_Logic_Vector(1 downto 0);
  variable B_v : Std_Logic_Vector(1 downto 0);
begin
  X_v := Some_Function(X);
  B_v := Some_Function2(B);
  A <= B_v;

  if X_v = "00" then
    C <= D;

  elsif X_v = "01" and G = "11" then
    C <= E;

  else
    C <= "00";
  end if;

  G <= B_v or D;
end process Better_Lbl;
end Redundancy_Better;

```

1. *Some\_Function* is used once (enhances efficiency)
2. *A* is assigned once (enhances readability)

**Figure 9.1.2.5-2 Redundant Code Avoidance (potpouri\redund2.vhd)**

#### 9.1.2.6 Avoid Unnecessary Processes

**Tip** Combine processes when possible. For example, all clocked register updates can be put into a single process.

### 9.1.3 Types

Numerical data types, such as *Integers*, execute faster than vector types (e.g., *Std\_Logic* and *Std\_Logic\_Vector* and *Bit\_Vector*), and should be used for arithmetic calculations when the number of bits does not exceed 31. Note that some simulators and synthesizers

will accept a 32 bit number, but this produces non-portable code (see section 1.5.1). When bit-field information is used, then integers can be costly because of the conversions functions required. There is no direct bit field manipulation of the *Integer* type. The *Integer* object has to be converted to an *array* (or vector) for proper operation. A trade-off should be performed between the cost for performing type conversions between *Std\_Logic\_Vector* and *Integer*, and subsequent calculations using *Integer*, or directly performing calculations on *Std\_Logic\_Vector*. It was observed that the time required for converting *Std\_Logic\_Vector* signal to an *Integer* variable, adding two *Integer* variables, and converting the results back to *Std\_Logic\_Vector* variable is faster than making an addition between two *Std\_Logic\_Vector* signals.

**Rule** Type conversions should be done only when the value is necessary. Avoid using type conversion without examining its potential usage.

**Rule** Enumerated types have better simulation speeds than constrained array types, since range checking is performed statically and not during the simulation.

**Rule** When modeling memories, type *Integer* requires less storage than an enumeration data type. For example an *Integer* typically requires four bytes of storage, whereas an enumeration type (e.g., *Bit*, *Std\_Logic*) requires one byte per enumeration.

#### 9.1.4 Constants

**Rule** Constants provide a very efficient method to replace type conversion functions, and are very useful as table lookups. Figure 9.1.4 demonstrates this concept. The example makes use of table lookups, attributes, and functions. Note that two coding styles are used in the body of the functions: one with the *if* clause, and the other with the *case* clause. The *case* clause is more efficient in terms of simulation, but the constant table lookup method is the most efficient because no conversion functions are called and executed.

```
entity Const is
end Const;

architecture Const_a of Const is
  type State_Typ is (Idle, Busy, DMA, HOLD, XFR);
  subtype Int0to4_Typ is Integer range 0 to 4;
  type Int2State_Typ is array(0 to 4) of State_Typ;
  type State2Int_Typ is array(State_Typ) of Int0to4_Typ;
  constant Int2State_c : Int2State_Typ :=
    (0 => Idle,
     1 => Busy,
     2 => DMA,
     3 => HOLD,
     4 => XFR);

  constant State2Int_c : State2Int_Typ :=
    (Idle => 0,
     Busy => 1,
     DMA => 2,
     HOLD => 3,
     XFR => 4);
```


**Rule** Constants to be used for type conversion (or table lookup)

```

function Int2State(Int : Integer) return State_typ is
begin
  case Int is
    when 0      => return Idle;
    when 1      => return Busy; ← Conversion function with the case
    when 2      => return DMA;
    when 3      => return HOLD;
    when 4      => return XFR;
    when others => return Idle;
  end case;
end Int2State;

function State2Int(State : State_Typ) return Int0to4_t typ is
begin
  if      State = Idle then return 0; ← Conversion function with the if
  elsif   State = Busy then return 1 ;
  elsif   State = DMA  then return 2;
  elsif   State = HOLD then return 3;
  else      return 4;
  end if;
end State2Int;

begin
  Test_Lbl: process
    variable Int_v : Int0to4_Typ := 3;
    variable State_v : State_Typ;
  begin
    State_v := Int2State_c(Int_v);      -- table lookup
    Int_v   := State2Int_c(State_v) - 1; -- table lookup

    State_v := State_Typ'val(Int_v);    -- attribute function
    Int_v   := State_Typ'pos(State_v);  -- attribute function

    State_v := Int2State(Int_v + 2);    -- function call
    Int_v   := State2Int(State_v) - 3;  -- function call
    wait;
  end process Test_Lbl;
end Const_a;

```

• Table Lookup is FASTEST  
 • Attribute function is next fastest, if optimized  
 • Function call with case statement is next  
 • Function call with if statement is slowest

The diagram shows a vertical flow of code. It starts with two conversion functions: 'Int2State' and 'State2Int'. Arrows point from these functions to a box containing a bulleted list of conversion methods. From the 'State2Int' function, an arrow points down to a process labeled 'Test\_Lbl'. Inside this process, there are four assignments: three using attribute functions ('State\_v := State\_Typ'val(Int\_v);', 'Int\_v := State\_Typ'pos(State\_v);') and one using a function call ('State\_v := Int2State(Int\_v + 2);'). An arrow points from the final assignment in the process back up to the bulleted list.

**Figure 9.1.4 Use of Constants for Type Conversions to Enhance Simulation Speed (potpouri\const.vhd)**

### 9.1.5 Files

Access to files and TextIO is generally considered a slow process, and should be avoided when possible. Methods to avoid file access, or minimize the heavy use of files include the following: 

1. **Use algorithms in VHDL to generate data.** LFSRs (discussed in section 5.4) are useful for the generation of pseudo-random data (see section 8.6 for an example). Dynamically generated data can sometimes be used instead of files.
2. **Use VHDL processes or VHDL components to supply instructions to a BFM.** The instruction sequence can be directly modeled in VHDL (see section 8.5). Reading coded instructions from a file is significantly slower because:

- TextIO is used to read lines from files.
  - Subprograms are needed to parse each line into various fields and types (e.g., instructions and data fields).
  - A process is required to interpret the parsed instructions, and to provide the necessary actions in the model.
3. **Use data reduction to avoid the need to write into files.** Data reduction methods are more efficient than the traditional method of storing simulation results in files, and performing post processing on those files (e.g., data comparisons, data analyses). Two commonly used data reductions techniques include:
- Use of a signature register(s). A signature register provides a signature over a set of signals. A MISR placed in the testbench, external to the UUT, provides a means to verify that all outputs of the UUT are identical to a simulation known to be correct. The data should be strobed when it is stable. (Refer to section 5.3 for a definition of a MISR package, section 8.3 for a discussion of regression test, and section 8.6 for an application example).
  - Use of a verifier. A verifier is a VHDL model (e.g., components or processes) that verifies that the UUT meets the specifications. Errors include functional errors (such as protocol, or data errors), and timing errors. The verifier can log all errors in a file and/or on the screen. This technique of on-the-fly verification versus post processing verification provides several advantages:
    - \* Errors are discovered sooner in the verification process.
    - \* Once a serious error is discovered, the simulation can be stopped (with the *assert* statement and severity *failure*), thus avoiding the need to continue simulation.
    - \* Simulation speed is enhanced because of lack of TextIO processing and type conversions. Cost (simulation time) of verification in VHDL, outweighs the time for post processing of the files generated during simulation.
4. **Read file once if contents are re-used.** If a file is read several times (opened and closed) then it would be more efficient to read this file once, and to store its contents into a data structure. If the size of the data file is deterministic, then a fixed array can be used. Otherwise, a linked-list approach is necessary. If the contents of the file is formatted (e.g., contains fields of type *Integer*, *Std\_Logic\_Vector*), then the conversions (from Text) should be done prior to the storing of data into the data structure (since this conversion would be done once).
5. **Read and write into files using the binary format.** Binary format is much faster than ASCII text because conversion functions and TextIO routines are not required. A record can be used to collect a set of data (the file type mark can be of a record type). The model can then write multiple records into a file(s) in a binary format. This approach is useful when the data generated (i.e., written) by one model is later used (or read) by another model.

If a file is used through many simulation iterations, it may be beneficial to convert this text file into a binary file using a simple VHDL program. The model can then make use of the binary file. Figure 9.1.5-1 presents an example for writing data of a record type. Figure 9.1.5-2 presents an example for reading such data. Files *potpouri\finta87.vhd* and *potpouri\finb87.vhd* (supplied on disk) represent the VHDL'87 versions of the same examples. Note that the binary format may not be portable between platforms and/or simulators. Therefore, once a tool and a platform are defined, text files that will be reused should first be translated to binary format.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
package Types_Pkg is
    type DataRcd_Typ is record
        Data32 : Std_Logic_Vector(31 downto 0);
        Data16 : Std_Logic_Vector(15 downto 0);
        DataInt : Integer;
        DataReal : Real;
    end record;
end Types_Pkg;

library IEEE;
use IEEE.Std_Logic_1164.all;
library work;
use Work.Types_Pkg.all;
entity FileWr is
end FileWr;

architecture FileWrRcd93_a of FileWr is
    subtype Std32_Typ is Std_Logic_Vector(31 downto 0);
begin -- FileWr_a
    WriteFile_Lbl : process
        type Stdfile_Typ is file of Work.Types_Pkg.DataRcd_Typ;
        file DataOut_f : Stdfile_Typ open Write_Mode is "IntOut.bin";
        variable Std32_v : Std32_Typ := 
            "UX01LWHZ-1011000001111110000111";
        variable fStatus_v : File_Open_Status;
        variable Int_v : Integer := 5;
        variable Real_v : Real := 3.14;
        variable Rcd_v : Work.Types_Pkg.DataRcd_Typ;
    begin -- process WriteFile_Lbl
        Write_Lbl: for Count_i in 1 to 10 loop
            Std32_v := Std32_v(30 downto 0) & Std32_v(31);
            Int_v := Int_v + 1;
            Real_v := Real_v + 1.03;
            Rcd_v.Data32 := Std32_v;
            Rcd_v.Data16 := Std32_v(15 downto 0);
            Rcd_v.DataInt := Int_v;
            Rcd_v.DataReal := Real_v;
            Write(F      => DataOut_f,
                  Value   => Rcd_v);
        end loop Write_Lbl;
        wait;
    end process WriteFile_Lbl;
end FileWrRcd93_a;

```

**Figure 9.1.5-1 Write Binary file (VHDL'93) (*potpouri\finta93.vhd*)**

```
library IEEE;
  use IEEE.Std_Lock_1164.all;
library work;
  use Work.Types_Pkg.all;

entity FileRead is
end FileRead;

architecture FileRead93_a of FileRead is
  subtype Std32_Typ is Std_Lock_Vector(31 downto 0);
  signal Std32_s : Std32_Typ;
begin -- File_a
  ReadFile_Lbl : process
    type StdFile_Typ is file of Work.Types_Pkg.DataRcrd_Typ;
    file DataIn_f : StdFile_Typ;
    variable fStatus_v : File_Open_Status;
    variable Data32_v : Std_Lock_Vector(31 downto 0);
    variable Data16_v : Std_Lock_Vector(15 downto 0);
    variable DataInt_v : Integer;
    variable DataReal_v : Real;
    variable Rcd_v : Work.Types_Pkg.DataRcrd_Typ;

  begin -- process ReadFile_Lbl
    File_Open(Status      => fStatus_v,
              F          => DataIn_f,
              External_Name => "IntOut.bin",
              Open_Kind   => Read_Mode);
    while not Endfile(DataIn_f) loop
      Read(F      => DataIn_f,
            Value  => Rcd_v);
      Std32_s <= Rcd_v.Data32;
      Data32_v := Rcd_v.Data32;
      Data16_v := Rcd_v.Data16;
      DataInt_v := Rcd_v.DataInt;
      DataReal_v := Rcd_v.DataReal;
      wait for 10 ns;
    end loop;
    wait;
  end process ReadFile_Lbl;
end FileRead93_a;
```

Figure 9.5-2 Read Binary file (VHDL'93) (potpourri\fintb93.vhd)

The syntax for a file declaration is shown below:

```
-- VHDL'87
file_declaration ::= 
    file identifier : subtype_indication is [ mode ]
        file_logical_name ;
-- VHDL'93
file_declaration ::= 
    file identifier_list : subtype_indication
        file_open_information ] ;

subtype_indication ::= 
    [ resolution_function_name ] type_mark
    [ constraint ]
```

Note that VHDL has some strict rules about type mark definition of a file, and includes the following:

1. *[1] The type mark may denote either a constrained or unconstrained type.*
2. *The base type of this subtype must not be a file type or an access type.*
3. *If the base type is a composite type, it must not contain a subelement of an access type.*
4. *If the base type is an array type, it must be one dimensional array type.*

### 9.1.6 Subprograms

**Rule** Avoid passing large data structures into subprograms, since they may be passed by value or copy. Consider using in-line code, or repartition the problem by using processes that have visibility into the signals and ports of the architecture.

Avoid side effects in subprograms for reasons explained in section 4.1.

### 9.1.7 Memory

Simulation performance is greatly impacted by the size and distribution of frequently accessed VHDL model memory locations representing objects. The host architecture, cache size, and implementation significantly impact the performance. However, the size of the objects being modeled (e.g., variable, signal) may also impact the simulation performance. For example, the model of a RAM requires a very large amount of storage in the host. A RAM with a 32-bit address space and a 32-bit width requires  $2^{32} \times 32$  Bytes, or 138.4 GBytes when the RAM is modeled with a variable. On a personal computer system, declaring a variable that requires a large amount of storage can cause an “out of memory” system fault. On a workstation, a similar crash can occur if the amount of memory required by the simulator exceeds the capabilities of the workstation. A simple model, with a variable declaration requiring one GByte of storage on the workstation (file *potpourri\crash.vhd*, architecture *EIM\_a*), brought a *SUN SPARCstation* to a grinding halt. That workstation was operating the *SunOS 4.1.3\_UI*, and had 512 MBytes of memory, with 1.1 GBytes of swap space.

In practical simulations, not all of the RAM address space is utilized. If this is the case, then a dynamic allocation of memory pages using linked lists can be used. Thus, memory

is allocated only for those pages that are being addressed (see section 6.1.5).

Another method to conserve host memory space is to avoid memory leaks (see section 5.1).

### 9.1.8 Packages

Many tool vendors either accelerate or optimize the following packages:

- *Std\_Logic\_Arith*
- *Std\_Logic\_Unsigned*
- *Std\_Logic\_Signed*
- *Numeric\_Std*
- *Numeric\_Bit*
- *Vital\_timing* (both 2.2b and 3.0)
- *Vital\_primitives* (both 2.2b and 3.0)

The use of these accelerated packages will result in faster simulations. Therefore, a user is encouraged to use these packages instead of attempting to write another version of equivalent packages.

### 9.1.9 VITAL

VITAL primitives are accelerated in many simulators. To speed up simulation of designs using VITAL models, the following controls are recommended:

- Turn off the timing checks generic unless timing checks are needed.
- Turn off glitch messages.
- Turn off X-generation, unless needed.

## 9.2 ACCESSING SIGNALS INTERNAL TO COMPONENTS

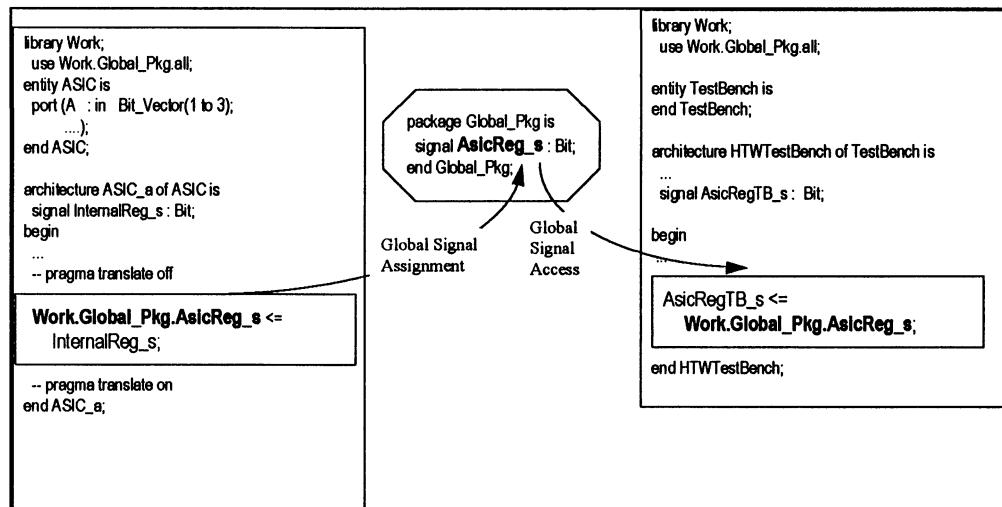
**Q** How can signals internal to a component be accessed in VHDL? One specific application is the generation of trace files of ASIC simulations. Another application is the use of internal signals by the testbench (such as driving a load on a signal when the ASIC is in tri-state mode). VHDL does not allow access of internal signals within a component. How can this be performed from within VHDL code in a testbench architecture that instantiates the component under test?

**A** There are two methods to access signals internal to a component:

1. Global signal method
2. Port method

### 9.2.1 Global Signal Method

Global signals are signals declared in packages. Those signals are made visible in any architecture that makes the library accessible to the architecture. The desired internal signals of an architecture can be assigned to the global signals. A testbench can then access those global signals and store the data into text files. This concept is graphically demonstrated in figure 9.2.1-1.



**Figure 9.2.1 Use of Global Signals to Transfer Information to other Components**

Global signal assignments in an architecture enable other components to examine the values asserted on those signals. Pragmas (or directives) must be used to direct the compiler to ignore global signals and global signal assignments.

One of the drawbacks of this solution is that it requires modifying the VHDL model. Users may feel that editing a VHDL model (to include the global signal assignments) is a poor design methodology, and may be error prone.

Figure 9.2.1-2 is a model of a three-bit counter that outputs the value of the count when the LSB of Count equals '1'. Otherwise, the output is tri-stated. Access of the internal tri-state control signal for this output is provided through a global signal.

```

library IEEE;
use IEEE.Std_Lock_1164.all;

package My_Pkg is -- assume compilation in library My_Lib
    signal DataTriCntrl : Std_uLogic;
end My_Pkg ;

library IEEE;
use IEEE.Std_Lock_1164.all;
use IEEE.std_logic_arith.all;

-- pragma translate off ←
library Work;
use Work.My_Pkg; --
-- pragma translate on

entity Cntlr is
    -- 3-bit counter (Count) which synchronously is reset to
    -- "000" when Reset = '1'.
    -- Data equals Count when count(0) equals '1', else
    -- Data equals Count when count(1) equals 'Z'
    -- Must trace the output enable control signal.
    port(Reset      : in Std_uLogic;
          Clk        : in Std_uLogic;
          Data       : out Std_Lock_Vector(2 downto 0));
end Cntlr;

architecture Cntlr_a of Cntlr is
    signal Count_s : Std_Lock_Vector(2 downto 0);
begin
    Count_lbl: process
    begin
        wait until Clk'event and Clk = '1';
        if Reset = '1' then
            Count_s <= "000";
        else
            Count_s <= Unsigned(Count_s) + 1;
        end if;
    end process Count_Lbl;

    -- Concurrent signal assignment
    Data <= Count_s when Count_s(0) = '1' else
        "ZZZ";

    -- pragma translate off ← Global signal assignment
    My_Pkg.DataTriCntrl <= Count_s(0);
    -- pragma translate on
end Cntlr_a;

```

Global signal assignment. Pragmas inform  
synthesizer to ignore code braced within the pragmas

Type conversion to *Unsigned*.  
This overloaded "+" operator  
limits results size of parameters.

**Figure 9.2.1-2 Use of Global Signals to Access Signals Internal to Components (potpourri\hierglob.vhd)**

Figure 9.2.1-3 represents a testbench that instantiates the test component. This testbench generates to the test output file a trace of the desired output enable signal (internal to a component).

```

library IEEE;
use IEEE.Std_Logic_1164.all;
library Work;
use Work.My_Pkg; --
entity TestBench is
end TestBench;

architecture TestBench_a of TestBench is
component Cntlr
port(Reset : in Std_uLogic;
      Clk     : in Std_uLogic;
      Data    : out Std_Logic_Vector(2 downto 0));
end component;

signal Reset : Std_uLogic := '0';
signal Clk   : Std_uLogic := '0';
signal Data  : Std_Logic_Vector(2 downto 0);
begin
U1_Cntlr: Cntlr
port map
(Reset => Reset,
 Clk     => Clk,
 Data    => Data);

Trace_Lbl: process(Clk)
use Std.TextIO.all;
use Std.TextIO;
use IEEE.Std_Logic_TextIO.all;
use IEEE.Std_Logic_TextIO;
file DataOut_f : TextIO.Text is out "SimOut.txt";
variable OutLine_v : TextIO.Line;
begin
if Clk'event and Clk = '0' then -- falling edge
  TextIO.Write(Outline_v, now);
  TextIO.Write(Outline_v, string'(" "));
  Std_Logic_TextIO.Write(Outline_v, My_Pkg.DataTriCntrl);
  TextIO.Write(Outline_v, string'(" "));
  Std_Logic_TextIO.Write(Outline_v, Reset);
  TextIO.Write(Outline_v, string'(" "));
  Std_Logic_TextIO.Write(Outline_v, Data);
  TextIO.WriteLine(DataOut_f, Outline_v);
end if;
end process Trace_Lbl;

Clk <= not Clk after 50 ns;

Reset <= '0' after 0 ns,
      '1' after 100 ns,
      '0' after 300 ns;
end TestBench_a;

```

Global signal provides access to internal value of component

**Figure 9.2.1-3 Testbench for Access of Component Internal Signals  
(potpouri\hierglob.vhd)**

### 9.2.2 PORT Method

 This approach adds ports to the model to provide visibility into signals internal to the architecture. Thus, the interface is entirely expressed by the entity. Pragmas can be used to prevent the synthesizer from compiling these ports.

This approach can cause confusion among readers because the entity includes additional ports declared (and bracketed by pragmas), but not representing physical pins of the implemented component.

Figure 9.2.2-1 demonstrates the port method concept. This model is identical in performance to the previous model. The test port is used to provide visibility into the tri-state control signal for this output.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

entity Cntlr is
  -- Must trace the output enable control signal.
  port(Reset      : in Std_Logic;
        Clk       : in Std_Logic;
        Data      : out Std_Logic_Vector(2 downto 0);
        -- pragma translate off
        DataTriCntrl : out Std_Logic  ← Phantom port not used by synthesizer.
        -- pragma translate on
        );
end Cntlr;

architecture Cntlr_a of Cntlr is
  signal Count_s : Std_Logic_Vector(2 downto 0));
begin
  Count_lbl: process
  begin
    wait until Clk'event and Clk = '1';
    if Reset = '1' then
      Count_s <= "000";
    else
      case Count_s is
        when "000" => Count_s <= "001";
        ...
        when others => Count_s <= "000";
      end case;
    end process Count_Lbl;

    -- Concurrent signal assignment
    Data <= Count_s when Count_s(0) = '1' else "ZZZ";
    -- pragma translate off ← Phantom port assignment.
    DataTriCntrl <= Count_s(0);  -- '0' disables data
    -- pragma translate on
  end Cntlr_a;

```

**Figure 9.2.2-1 Use of Ports to Provide Visibility Into Signals Internal to Component (potpuri\hierport.vhd)**

Figure 9.2.2-2 represents a testbench with the port method to provide visibility into signals internal to components.

```

library IEEE;
use IEEE.Std_Logic_1164.all;

entity TestBench is
end TestBench;

architecture TestBench_a of TestBench is
component Cntlr
port(Reset      : in Std_uLogic;
      Clk        : in Std_uLogic;
      Data       : out Std_Logic_Vector(2 downto 0);
      -- pragma translate off
      DataTriCntrl : out Std_Logic
      -- pragma translate on
    );
end component;
signal Reset      : Std_uLogic := '0';
signal Clk        : Std_uLogic := '0';
signal Data       : Std_Logic_Vector(2 downto 0);
signal DataTriCntrl : Std_uLogic := '0';
begin
  U1_Cntlr: Cntlr
  port map
    (Reset      => Reset,
     Clk        => Clk,
     Data       => Data,
     DataTriCntrl => DataTriCntrl);

  Trace_Lbl: process(Clk)
    use Std.TextIO.all;  use Std.TextIO;
    use IEEE.Std_Logic_TextIO.all;  use IEEE.Std_Logic_TextIO;
    file DataOut_f : TextIO.Text is out "SimOut.txt";
    variable OutLine_v : TextIO.Line;
  begin
    if Clk'event and Clk = '0' then -- falling edge
      Std_Logic_TextIO.Write(OutLine_v, DataTriCntrl);
      TextIO.Write(Outline, " ");
      Std_Logic_TextIO.Write(OutLine_v, Reset);
      TextIO.Write(Outline, " ");
      Std_Logic_TextIO.Write(OutLine_v, Data);
      TextIO.WriteLine(DataOut_f, Outline);
    end if;
  end process Trace_Lbl;
  Clk  <= not Clk after 50 ns;

  Reset <= '0' after 0 ns,
         '1' after 100 ns,
         '0' after 300 ns;
end TestBench_a

```

**Figure 9.2.2-2 Testbench with Ports to Provide Visibility Into Signals Internal to Component (potpuri\hierport.vhd)**

### 9.3 TRANSFERRING A LINE ONTO A SIGNAL

**Q**

VHDL does not allow a signal to be of *access* type. How can an object of type *Std.TextIO.Line* be transferred from one component to another?

**A**

In VHDL'93 shared variables can be used to define an object of *access* type. Shared variables can also be declared in packages, and made visible to all components. In VHDL'87 the contents of the line (i.e., a constrained string) can be used for the transfer through a global signal or a port. The constrained string must be long enough to hold the contents of a line. A function is required to either extend or clamp the contents of a read line to the constrained string. Figure 9.3-1 demonstrates the concept with a *shared* variable. Figure 9.3-2 demonstrates the use of a signal. Note that this signal can be a signal internal to the architecture, a port, or a global signal.

```
-- Title      : Demonstration of Line transfer with VHDL'93
-- Description : Read data from a file, send line to another
--                : process, and write it to the screen
--                : and to an output file
library STD;
use Std.TextIO.all;
use Std.TextIO;
entity SharedLine is
end SharedLine;

architecture SharedLine93_a of SharedLine is
shared variable LineShared_v  : Std.TextIO.Line;
signal DataRdy_s : Boolean;
signal EOT_s     : Boolean;
begin -- SharedLine_a
ReadFile_Lbl : process
file DataIn_f      : TextIO.Text open Read_Mode  is
  "datain.txt";
variable InLine_v   : TextIO.Line; -- pointer to string

begin
  while not TextIO.Endfile(DataIn_f) loop
    -- Read 1 line from the input file
    TextIO.ReadLine(DataIn_f, InLine_v);
    LineShared_v := InLine_v;
    DataRdy_s <= True;
    wait on EOT_s'transaction;

    -- If here, then other process is done with the line.
    -- Other process also deallocate the storage.
  end loop;
  wait;
end process ReadFile_Lbl;
```

```

WriteFile_Lbl : process
  file DataOut_f : TextIO.text open Write_Mode is "dataout.txt";
  variable OutLine_v : TextIO.Line; -- pointer to string

begin -- process ReadWriteFile_Lbl
  wait on DataRdy_s'transaction;
  -- Create a new pointer to point to a new string
  -- so that the new line can be used for OUTPUT to transcript
  -- without affecting the read line. Note that a WriteLine
  -- deallocates the line, and the data is deleted.
  OutLine_v := new String(LineShared_v'range)'(LineShared_v.all);

  -- Write to Output and to file
  TextIO.WriteLine(Output, OutLine_v); -- to screen
  TextIO.WriteLine(DataOut_f, LineShared_v);
  EOT_s <= True;
end process WriteFile_Lbl;
end SharedLine93_a;

```

**Figure 9.3-1 Transfer of Line with Shared Variables using VHDL'93  
(potpouril\line93.vhd)**

```

library STD;
  use Std.TextIO.all;  use Std.TextIO;
entity StringLine is
  generic(MaxWidth_g : Natural := 80);
end StringLine;

architecture StringLine87_a of StringLine is
  signal String_s : String(1 to MaxWidth_g);
  signal DataRdy_s : Boolean;
  signal EOT_s : Boolean;
begin -- StringLine_a
  ReadFile_Lbl : process
    file DataIn_f : TextIO.text is "datain.txt";
    variable InLine_v : TextIO.Line; -- pointer to string
    variable Index_v : Natural;
    subtype StringSized_Typ is String(1 to MaxWidth_g);

    function FixedString(StrgIn_c : String) return
      StringSized_Typ is
      variable S_v : StringSized_Typ := (others => ' ');
    begin
      if StrgIn_c'length <= MaxWidth_g then
        S_v(1 to StrgIn_c'length) := StrgIn_c;
        return S_v;
      else
        return StrgIn_c(1 to MaxWidth_g);
      end if;
    end FixedString;

```

```

begin -- process ReadWriteFile_Lbl
  while not TextIO.Endfile(DataIn_f) loop
    -- Read 1 line from the input file
    TextIO.ReadLine(DataIn_f, InLine_v);
    String_s <= FixedString(InLine_v.all);
    deallocate(InLine_v);
    wait for 1 ns;
  end loop;
  wait;
end process ReadFile_Lbl;

WriteFile_Lbl : process
  file DataOut_f      : TextIO.text is out "dataout0.txt";
  variable OutLine_v  : TextIO.Line; -- pointer to string
  variable OutLine2_v : TextIO.Line; -- pointer to string
begin -- process ReadWriteFile_Lbl
  wait on String_s'transaction;
  -- Create a new pointer to point to a new string
  -- so that the new line can be used for OUTPUT to transcript
  -- without affecting the read line. Note that a WriteLine
  -- deallocates the line, and the data is deleted.
  OutLine_v := new String(String_s'range)'(String_s);
  OutLine2_v := new String(String_s'range)'(String_s);
  TextIO.WriteLine(Output, OutLine_v);           -- to screen
  TextIO.WriteLine(DataOut_f, OutLine2_v);        -- to file
end process WriteFile_Lbl;
end StringLine87_a;

```

Line is converted to a fixed string of characters  
and assigned onto a signal of the proper type

**Figure 9.3-2 Transfer of Line with Signal Using VHDL'87  
(potpourri\line87.vhd)**

#### 9.4 TYPE DECLARATION IN MULTIPLE PACKAGES

**Q**

The following type is declared twice, once in a package, and once in an architecture declaration section of a testbench:

type TimeArray\_Typ is array (natural range <>) of time;

The entity that uses a port of this type compiles with no errors. However, when the component is instantiated in a testbench (with the redeclaration of type *TimeArray\_Typ*) the compiler gives the following error for the testbench architecture:

\*\*Error: The type of the actual in argument 6 does not  
match the type of the corresponding formal.

Why does the compiler yield an error?

**A**

The problem is that the code has two different type declarations (even though they have the same declarations). One declaration is defined in the package used by the component, and another one is defined in a separate package used by the testbench. VHDL, unlike Pascal, does not have "type isomorphism." To solve the problem, only one of the declarations should be used everywhere. The type declaration should be defined in a package in this case.

## 9.5 INTERNET - FREQUENTLY ASKED QUESTIONS

**Q** | Where can general VHDL information about books, tools, products and services be found on the Internet?

**A** | There are Frequently Asked Questions (FAQ) listings, updated monthly, on the Internet. They can be accessed from the following sites:

From near Europe:

<http://lsl-www.informatik.uni-dortmund.de/~dettmer/>

From near USA

<http://vhdl.org/vi/comp.lang.vhdl/>

<ftp://vhdl.org/vi/comp.lang.vhdl/>

At the time of this writing, there were four regular postings. These FAQ listings, as of the publication date of this book, are available in the *potpourri* subdirectory in files FAQ1.txt, FAQ2.txt, FAQ3.txt, FAQ4.txt.

## 9.6 VHDL TEXT EDITOR?

**Q** | What is a good VHDL text editor?

**A** | The best VHDL language sensitive editor (LSE) is *emacs*.

1. *Emacs* is a true LSE editor, with pre-built templates for all the VHDL constructs. VHDL mode also provides the abbreviation mode to allow the quick recall of all typed strings (e.g., signal names) without having to retype them. This introduces consistency and accuracy in the entry of VHDL code. In addition, the user is relieved of the duty to remember the syntax. All reserved words are highlighted.
2. *Emacs* is a very powerful editor in its own right with many features including multi-frames.
3. *Emacs* runs on several platforms, including *Suns*, *HPs*, and *PCs* thus making the transition between platform transparent.
4. *Emacs* is GNU software, and thus is freely available.
5. Many other tool editors are based on *emacs*, thus easing the transition to other tools.

*Emacs* may be difficult to learn; but once mastered, it is extremely powerful, flexible, and indispensable as a text editor.

The current official *emacs* VHDL ftp repository is at <ftp://ftp.eda.com.au/pub/emacs/>. The VHDL mode file is *vhdl-mode.tar.gz* (see FAQ1 listing, section 9.5).

Another powerful editor based on *emacs* is *lemacs* that is Cadence's *Leapfrog®* editor. This editor has an enhanced graphical user interface and is well integrated with the

compiler. *Lemacs* is distributed with *Leapfrog®*, and is available for workstations only.

For engineers running Windows and looking for an alternative to *emacs*, there are some Windows based LSE editors available. The reader should refer to the FAQ 3 listing (Section 9.5) for a list of available products.

## 9.7 VITAL

**Q** | What is *VITAL*?

**A** | The VHDL Initiative Towards ASIC Libraries (VITAL) is an industry-based, informal consortium formed with the following in mind:

- **Charter:** Accelerate the availability of ASIC libraries across industry VHDL simulators.
- **Objective:** High-performance, accurate (sign-off quality) ASIC simulation across VITAL-compliant Electronic Design Automation (EDA) tools from a single ASIC vendor description.
- **Approach:** Define a modeling specification (in conjunction with VHDL packages) that leverages existing practices and techniques, is compliant with IEEE Standards 1076 and 1164, and utilizes *Open Verilog International's* (OVI) and *Standard Delay Format* (SDF) timing format. Standardize the approved result through the IEEE.
- **End-Product:**
  - (1) *VITAL\_Timing* VHDL package defining standard, acceleratable timing procedures for delay value selection, timing checks, and timing error reporting;
  - (2) *VITAL\_Primitives* VHDL package defining standard, acceleratable primitives for Boolean and table-based functional description;
  - (3) Specification of OVI's Standard Delay Format (SDF) for communication of instance delay values; and,
  - (4) Model Development Specification document defining utilization of VITAL and VHDL elements for ASIC library development.

### 9.7.1 VITAL Features

The main features of VITAL (based on version 2.2B) include the following:

1. **Modeling specification.** This specification covers:
  - **Naming conventions** for the timing parameters and internal signals.
  - **Use of types** defined in the timing package to specify timing parameters.
  - **Coding style methodology** to define and back annotate timing parameters.
  - **Two Coding styles** that accommodate acceleratable models:

- . Pin-to-pin delay style with a single process to describe the behavior.
  - . Distributed delay style with use of predefined concurrent procedures.
- Two levels of compliance:
    - Level 0: Complex models described at higher level,
    - Level 1: Model acceleration permitted.
2. **VITAL timing package.** This package contains data types and subprograms to support development of macrocell timing models. Included in this package are routines for delay selections, timing violations checking and reporting, and glitch detection.
  3. **VITAL\_Primitives package.** This package is a set of commonly used combinatorial primitives provided both in Function and concurrent Procedure form to support either behavioral or structural modeling styles. Examples of VITAL primitives include *VitalAND*, *VitalOR*, *Vital AND2*, *VitalMux4*, etc.). The procedure primitives support separate pin-to-pin delay path and *GlitchOnEvent* glitch detection. In addition, this package contains general purpose Truth and State Tables that are very useful in defining state machines and registers.
  4. **VITAL\_SDFmap.** The Standard Delay File (SDF) to VHDL mapping specification defines the mapping and translation of SDF data to Generic Parameter values on VITAL models. Simulator vendors use this mapping standard to build tools that automatically back-annotate the VHDL circuit and timing data values.

VITAL coding styles, package descriptions, and specifications are described in a document entitled *VITAL Model Development Specification* that may be downloaded via ftp from the Internet site "vhdl.org" in subdirectory "/vi/vital". The FAQ 1 listing (section 9.5) provides Internet sites that have tutorials on many topics, including VITAL.

## 9.8 BEHAVIORAL MODELING

**Q** | What is behavioral modeling, and how does it differ from register transfer level (RTL) modeling?

**A** | Behavioral modeling represents a higher level of abstraction than RTL. It provides a high level of flexibility because ALL the programming language features can be used to specify a design. Thus, all the traditional synthesis language restrictions need not be complied. For example, in behavioral modeling, multiple wait statements in a process can be used to represent a sequence of operations occurring at different clock cycles or events. Another example is the emulation of multi-cycle operation (such as a multiply) with a straight multiplication and a time delay (e.g.,  $S \leq A * B$  after  $4 * \text{Cycle\_Delay}$ ).

RTL modeling is more restrictive than behavioral modeling, and defines all transactions for each clock cycle. It generally follows the synthesis restrictions, and is often synthesized into a gate level implementation with a synthesizer.

## 9.9 FINAL VHDL EXAM

This section provides a university type of final exam on VHDL. It also serves as a review of key VHDL concepts.

### 9.9.1 Design Units

**Q** | What are the five VHDL design units? What do they represent? Which of these units, or pairs of units can be simulated?

**A** | The five design units and their representations are summarized in table 9.9.1

Table 9.9.1 VHDL Design Units

Design Unit	Comments
1. ENTITY	It represents the interface specification, or inputs and outputs of a design. It is analogous to a symbol on a schematic.
2. ARCHITECTURE	It represents the functionality or composition of an entity. An architecture may be described at various levels including behavioral, RTL, or structural. Together the entity and architecture pair make up a simulatable component.
3. PACKAGE DECLARATION	It provides a collection of declarations (types, constants, deferred constants, signals, components) or subprogram declarations (procedures, functions). Shared variables (VHDL'93) can also be declared.
4. PACKAGE BODY	It provides definitions of the subprogram bodies (e.g., the algorithms). Values for deferred constants are also declared.
5. CONFIGURATION DECLARATION	It binds component instantiations to entity and architecture pairs.

The simulatable design units include:

- Entity and architecture pairs
- Configuration declarations

Packages are not simulatable.

### 9.9.2 Compilation Order

**Q** Identify which design units must be recompiled when there is a change in one of the following design units. Assume that the design makes use of a user defined package. Provide a rationale for your decision.

**Case 1:** Entity that represents a component instantiated in a higher level architectures.

**Case 2:** Entity that represents the highest level of the design.

**Case 3:** Architecture of a component.

**Case 4:** Package declaration.

**Case 5:** Package body.

**Case 6:** A configuration declaration used at higher level configurations.

**A** Each of the six cases is discussed below. The general recompilation rule is that any change to an interface item requires a recompilation of all items that use that interface. Any change to a body of an item requires recompilation of that body only.

**Case 1:** Entity that represents a component instantiated in a higher level architecture.

Design Unit to be recompiled	Rationale
Entity in question	Design change in entity
All Architectures of that entity	A change in the entity may include a change in a port, a generic, or other entity declarations. Since the architectures of that entity rely on those declarations, they must be recompiled.
All architectures that instantiate the modified component	A change in the entity may include a change in a port, a generic, or other declarations. Since the architectures that instantiate the component rely on those declarations, they must be recompiled.
All configurations that reference that entity	A change in the entity may include a change in a port, a generic, or other declarations. If a configuration references the entity it must be recompiled (whether or not it uses the port and generic mappings).
All configurations that reference a modified configuration	Dependency clause. The entity referenced in the modified configuration may have changed.

*Note:* The entities of the architectures that instantiate the modified component need not be recompiled. Therefore, if an entity  $X$  has an architecture  $X\_a$  that instantiates component  $Y$ , and the entity  $Y$  is modified, then  $X\_a$  must be recompiled. This is necessary because the  $Y$  interfaces may have been modified. However, entity  $X$  need not be recompiled since its interfaces were not modified.

**Case 2:** Entity that represents the highest level of the design.

Design Unit to be recompiled	Rationale
Entity in question	Design change in entity
the Architecture of that entity	A change in the entity may include a change in a port, a generic, or other declarations. Since the architecture of that entity relies on those declarations, it must be recompiled.
All configurations that reference that entity	A change in the entity may include a change in a port, a generic, or other declarations. If a configuration references the entity whether or not it uses the port and generic mappings, it must be recompiled.

**Case 3:** Architecture of a component.

Design Unit to be recompiled	Rationale
The modified architecture	Since the entity has not changed, only that architecture needs to be recompiled.

*Note:* No other design unit needs to be recompiled since the entity has not changed.

**Case 4:** Package declaration.

Design Unit to be recompiled	Rationale
Package declaration	Design change in the declaration.
All package declarations and package bodies that make use of the modified package declaration.	Dependency clause. The package declarations may depend on definitions defined in the modified package.
Package body of the modified package.	A change in the declaration requires a recompilation of the body because types, constants, subprogram interfaces may have been modified.
All Entities and architectures using the modified package.	Dependency clause.
All Architectures of the recompiled entities.	A change in the entity may include a change in a port, a generic, or other declarations. Since the architectures of that entity rely on those declarations, they must be recompiled.
All architectures that instantiate the recompiled components	A change in the entity may include a change in a port, a generic, or other declarations. Since the architectures that instantiate the component rely on those declarations, they must be recompiled.
All configurations that reference the recompiled entities.	A change in the entity may include a change in a port, a generic, or other declarations. If a configuration references the entity it must be recompiled (whether or not it uses the port and generic mappings).
All configurations that reference a modified configuration	Dependency clause. The entity referenced in the modified configuration may have changed.

### Case 5: Package body.

Design Unit to be recompiled	Rationale
Package body of the modified package.	Design change.

**Case 6:** A configuration declaration used at higher level configurations.

Design Unit to be recompiled	Rationale
The modified configuration.	Design change in the configuration declaration
All configurations that reference a modified configuration	Dependency clause. The entity referenced in the modified configuration may have changed.

### 9.9.3 VHDL Types

**Q** Provide in a package declaration an example of the following types or subtypes. In addition, define in the same package locally static constants that represent values for those types. Use named notation where applicable.

## Scalar

## Integer

## User defined enumeration

**Physical type for measures (i.e., meter, centimeter)**

## Composite

### Array (constrained and unconstrained)

## Record

### Access type (access to a record)

## File type

A

Figure 9.9.3 represents an example of such a package declaration.

```
package Types_Pkg is
    -- Scalar
    -- Integer
    type MyInt_Typ is range -5 to 7; -- Error: Declaration of integer subtype

    subtype Source_Typ is Integer range 31 downto 16;

    -- User defined enumeration
    type State_Typ is (Idle, Add, Sub, Pass, Lock);

    -- Physical type for measures (i.e., meter, centimeter)
    type Metric_Typ is range Integer'low to Integer'high
        units
            mm;           -- millimeter
            cm = 10 mm;   -- centimeter
            m = 100 cm;   -- meter
            km = 1000 m;  -- kilometer
    end units;

end Types_Pkg;
```

```
-- Composite
-- Array Unconstrained
type Astate_Typ is array(Integer range <>) of State_Typ;

-- Array Constrained
type Int2State_Typ is array(0 to 4) of State_Typ;
subtype MyInt2State_Typ is Astate_Typ(0 to 4); →
    

Avoid declaring  
multiple types. Use  
subtypes of  
unconstrained types.



-- Record
type Stuff_Typ is record
    Size : Metric_Typ;
    Numb : Integer;
    States : State_Typ;
end record;

-- Access type (to a record)
type Recrd_Typ;      -- incomplete type
type RecrdPntr_Typ is access Recrd_Typ;
type Recrd_Typ is record
    Stuff : Stuff_Typ;
    NextP : RecrdPntr_Typ;
end record;

-- File type
type BinFileFile_Typ is file of Stuff_Typ;

-- file declaration -- VHDL'93
file Bin_f : BinFileFile_Typ open Write_Mode is "BinOut.txt";

--Locally static constants
constant MyNumb_c : MyInt_Typ := 3;
constant Enable_c : Source_Typ := 30;
constant DoNothing_c : State_Typ := Idle;
constant BigLeap_c : Metric_Typ := 20 m;
constant Int2State_c : MyInt2State_Typ :=
    (0 => Idle,
     1 => Add,
     2 => Sub,
     3 => Pass,
     4 => Lock);

constant MyStuff_c : Stuff_Typ :=
    (Size => 10 m,
     Numb => 40,
     States => Pass);
end Types_Pkg;
```

Figure 9.9.3 Example of Type and Constant Declarations  
(potpourri\types\_p.vhd)

### 9.9.4 Attributes

**Q** What is an attribute? Identify the five classes of the predefined attributes. Why are predefined attributes important in VHDL?

**A** [1] An attribute is a value, function, type, range, signal, or a constant that may be associated with one or more names within a VHDL description. There are five classes of predefined attributes as shown in table 9.9.4.

Table 9.9.4 Classes of Predefined Attributes

Attribute class	Operation	Examples
Value	Returns a constant value	T'high, T'low, T'left, T'right, – T is a scalar type
Function	Calls a function that return a value	T'image, T'value, T'ascending – Scalar type T'pos, T'vel, T'succ, T'pred, T'leftof, T'rightof, – discrete type A'left, A'right, A'high, A'low, A'range, A'reverse_range, A'length, A'ascending – Array S'event, S'active, S'last_event, S'last_active, S'last_value – signal
Signal	Creates a new implicit signal	S'transaction, S'delayed, S'stable, S'quiet
Type	Returns a type	T'base
Range	Returns a range.	A'range, A'reverse_range

Attributes provide a mechanism to enhance the flexibility and reusability of a design. A change to an object parameter (such as its size, range, or length) can be easily accessed with attributes without a coding change. In subprograms, attributes of unconstrained arrays of formal parameters (such as *'length* or *'range*) can extract information (such as the size or range) about the actual parameters.

### 9.9.5 “WAIT” Statement

**Q** Given the following wait statements, identify for each of the following statements the sensitivity clause, condition clause, and the timeout clause:

Statement	sensitivity clause	condition clause	timeout clause
<b>wait until Clk = '1';</b>			
<b>wait on S until S2 = '1' and S = '0' for 15 us;</b>			
<b>wait until S = S2 for 10 ms;</b>			
<b>wait until S'active;</b>			
<b>wait on S'transaction for 50 ms;</b>			
<b>wait on S;</b>			
<b>wait until Var = 1;</b>			
<b>wait;</b>			

Note: S, S2, and Clk are signals, Var is a variable.

**A** Table 9.9.5 provides the sensitivity clause, condition clause, and the timeout clause for the defined expressions. Note that the expression *wait until S'active* is equivalent to *wait on S* (i.e., an event on *S*) because the implied sensitivity is on ALL the signals expressed in the condition clause. It is necessary to use the *wait on S'transaction* statement to achieve a sensitivity to any assignment on signal *S*. It is important to also note that the statement *wait until Var = 1* is equivalent to a *wait* because there is no signal expressed in the condition clause.

Table 9.9.5 Wait Expressions

Statement	sensitivity clause	condition clause	timeout clause
<i>wait until Clk = '1';</i>	<i>Clk</i>	<i>Clk = '1'</i>	<i>time'high</i>
<i>wait on S until S2 = '1' and S = '0' for 15 us;</i>	<i>S</i>	<i>S2 = '1' and S = '0'</i>	<i>15 us</i>
<i>wait until S = S2 for 10 ms;</i>	<i>S, S2</i>	<i>S=S2</i>	<i>10 ms</i>
<i>wait until S'active;</i>	<i>S</i>	<i>S'active</i>	<i>time'high</i>
<i>wait on S'transaction for 50 ms;</i>	<i>S'transaction</i>	<i>True</i>	<i>50 ms</i>
<i>wait on S;</i>	<i>S</i>	<i>True</i>	<i>time'high</i>
<i>wait until Var = 1;</i>	<i>none</i>	<i>True</i>	<i>time'high</i>
<i>wait;</i>	<i>none</i>	<i>True</i>	<i>time'high</i>

Note: *S*, *S2*, and *Clk* are signals, *Var* is a variable.

## 9.9.6 Control Structure

**Q** What techniques can be used to select discrete ranges of one dimensional arrays (e.g., *Std\_Logic\_Vector*) for use as loop control and aggregate choices?

**A** Various techniques are available in VHDL to define discrete ranges of one dimensional arrays. These include the following:

1. **Discrete values.** Specific discrete values are often used to identify the ranges that are not expected to be altered in future revisions of the design. Examples include:

```
for I in 10 downto 7 loop -- loop range
  S(10 downto 7) <= "1010"; -- slice
  S2(31 downto 11 => '1', -- aggregate
    10 downto 7 => '0', -- choices
    others       => 'L');
```

The use of discrete ranges does not provide flexibility in reusability or in making changes.

2. **Subtypes.** Integer subtypes that represent subfields of control registers can enhance code readability and reusability because the subtype name provides information about the intended functions. Subtypes are also more adaptable to design changes because a change in the subtype definition is automatically reflected whenever that subtype is used to identify a range. This rule applies provided the length of the vector that the subtype defines does not change. Examples include:

```
subtype Source_Typ is Integer range 10 downto 7;
subtype Dest_Typ  is Integer range 31 downto 11;

for I in Source_Typ loop -- loop range
  S(Source_Typ) <= "1010"; -- slice
  S2(Dest_Typ)   => '1', -- aggregate
  Source_Typ     => '0', -- choices
  others         => 'L');
```

The use of subtypes provide readability, adaptability to design changes, and reusability.

3. **Attributes.** Attributes can often be used to specify discrete ranges. Examples include:

```
for I in S1'range  loop -- loop range
for I in (S1'length - 1) downto 0  loop
for I in S'high to S'low  -- loop range
S(S1'range ) <= "1010";  -- slice
S2(S1'range  => '1',
S'range      => '0',  -- aggregate choices
others       => 'L');
```



The use of attributes  
enhance code reusability

### 9.9.7 Signals versus Variables

- Q** | How do signals compare to variables? What are the advantages of each? When do they imply registers, latches, and combinatorial logic? Given the following code, what are the final values of *S* and *V*?

```
architecture SigVar_a of SigVar is
  signal S : Integer := 10;

begin -- SigVar_a
  process
    variable V : Integer := 100;
  begin -- process
    S <= S + 5;
    V := S + 7;
    wait;
  end process;
end SigVar_a;
```

(potpouri\sigvar.vhd)

- A** | Signals can be defined in the following areas:
- Package declarative part of a package declaration (global signals),
  - Ports of an entity (component interfaces),
  - Architecture declarative part of an architecture (signals of the architecture),
  - Block declarative part of a block (local signals of a block),
  - Formal parameters of a subprogram (connected to actual signals during subprogram call).

A signal has three properties attached to it:

1. **Type** -- The type insures consistency in operations on objects.
2. **Value** -- This includes current, future, and past value (e.g. S'last\_value).
3. **Time** -- This represents a time associated with each value.

A variable can be defined in the following areas:

- Declarative part of a process,
- Declarative part of a subprogram,
- Shared variable (for VHDL'93) in the package declarative part of a package declaration,

- Shared variable (VHDL'93) in the architecture declarative part of an architecture,
- Shared variable (VHDL'93) in the architecture declarative part of a block.

A variable has two properties attached to it:

1. Type -- This is just like the signal properties, but there are no attributes associated with time.
2. Value – This represents the current value with no time history.

Since there is no time associated with variables, they are updated immediately, whereas signals are updated after a specified delay time. A zero delay time represents one delta time, or an infinitesimal small amount of time that is larger than zero. Thus, the ordering of signal assignments is not critical. It is however critical in variable assignments. For example, the following variable assignments yield:

$$\begin{array}{l} \text{A\_v := B\_v;} \\ \text{B\_v := A\_v;} \end{array} \quad \longrightarrow \quad \begin{array}{l} \text{A\_v gets value of B\_v} \\ \text{B\_v remains unchanged} \end{array}$$

The following signal assignments yield a true swap of signal values after one delta time. Signals change, or are assigned new values after one delta time (because delay is not specified, and is zero). Any signal assignment is made with the current value rather than the projected value, in one delta time.

$$\begin{array}{l} \text{A\_s <= B\_s;} \\ \text{B\_s <= A\_s;} \end{array} \quad \longrightarrow \quad \begin{array}{l} \text{A\_s gets value of B\_s} \\ \text{B\_s gets value of A\_s} \end{array}$$

Signals require a significant amount of storage overhead than variables. Signals typically require an additional 100 bytes per element (e.g., bit, integer). Thus, an array of 64,000 thirty-two bit words requires 2 MBytes ( $64\text{K} * 32 \text{ bits} * 1 \text{ byte/bit}$ ) for a variable declaration, and 207 MBytes ( $64\text{K} * 32 \text{ bits} * 101 \text{ byte/bit}$ ) for a signal declaration. This is very significant in the declaration of large data structures.

In synthesis, signals and variables can represent combinatorial logic or registers. See section 7.3 for a discussion of this topic.

The final values of the signal  $S$  in the above question is 15 after one delta time because it represents the addition of the current value of  $S$  plus the constant 5. The value of the variable  $V$  is 17 because it represents the current value of  $S$  plus the constant 7. Signal  $S$  changes in one delta time, unlike a variable that changes in zero time.

### 9.9.8 Operator Overloading

**Q** | The following types define complex numbers, where the *R* field represents the *real* number, and the *I* field represents the imaginary number. The *Signed* type is defined in the *Std\_Logic\_Arith* package.

```
type Complex32_Typ is record
    R : Signed(31 downto 0);
    I : Signed(31 downto 0);
end record;
type Complex64_Typ is record
    R : Signed(63 downto 0);
    I : Signed(63 downto 0);
end record;
```

Define in a package the following overloaded functions:

```
function "+" (A : Complex32_Typ;
              B : Complex32_Typ) return Complex32_Typ;

function "-" (A : Complex32_Typ;
              B : Complex32_Typ) return Complex32_Typ;

function "*" (A : Complex32_Typ;
              B : Complex32_Typ) return Complex64_Typ;
```

For complex numbers, the addition operators operate on the corresponding fields of the complex numbers. Thus, for the equation  $C := A + B$ , the following is true:

```
C.R := A.R + B.R;
C.I := A.I + B.I;
```

For multiplication ( $C := A * B$ ), the algorithm is demonstrated below:

```
C.R := (A.R * B.R) - (A.I * B.I);
C.I := (A.I * B.R) + (A.R * B.I);
```

**A** | See section 5.7 for a description of the package.

### 9.9.9 Model

**Q** | Write two models of a two-bit counter, one with the “+” operator, and the other without that operator. Write a testbench for this counter. Write configuration declarations that define the binding of the counter entity with each of the architectures.

**A** | The entity and architectures of the counter are shown in figure 9.9.9-1. The testbench is provided in figure 9.9.9-2. The configurations are included in figure 9.9.9-3.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_Unsigned.all;
entity Counter is
  port (Clk : in Std_Logic;
        Reset : in Std_Logic;
        Count : out Std_Logic_Vector(1 downto 0));
end Counter;

architecture Counter_a of Counter is
  signal Count_r : Std_Logic_Vector(1 downto 0); ← Package used for the "+" operator
begin -- Counter_a
  Count_Lbl : process
  begin -- process Count_Lbl
    wait until Clk'event and Clk = '1';
    if Reset = '1' then
      Count_r <= (others => '0');
    else
      Count_r <= Count_r + 1;
    end if;
  end process Count_Lbl;

  -- Concurrent statement
  Count <= Count_r; ← Assignment of registers to output.
end Counter_a;

architecture Counter_RTL of Counter is
  signal Count_r : Std_Logic_Vector(1 downto 0); ← Signal used for the register definition.
begin -- Counter_rtl
  Count_Lbl : process
  begin -- process Count_Lbl
    wait until Clk'event and Clk = '1';
    if Reset = '1' then
      Count_r <= (others => '0');
    else
      Count_r(0) <= not Count_r(0);
      if Count_r(0) = '1' then
        Count_r(1) <= not Count_r(1);
      end if;
    end if;
  end process Count_Lbl;

  -- Concurrent statement
  Count <= Count_r; ← Use of the "+" operator. Port Count cannot
end Counter_RTL; ← be used here because it is an output port, and
                     thus cannot be READ.
                     Combinatorial logic to implement the
                     increment function

```

Figure 9.9.9-1 Counter architectures (potpourri\cntr\_ea.vhd)

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity CntrTB is
begin
end CntrTB;

architecture CntrTB_a of CntrTB is
component Counter
    port(Clk      : in Std_Logic;
         Reset    : in Std_Logic;
         Count   : out Std_Logic_Vector(1 downto 0)
        );
end component;

signal Clk      : Std_Logic := '1';
signal Count   : Std_Logic_Vector(1 downto 0);
signal Reset   : Std_Logic;

begin
U1_Counter: Counter
port map(
    Clk      => Clk,
    Reset    => Reset,
    Count   => Count
);

Reset <= '1' after 205 ns,
          '0' after 305 ns;

Clk  <= not Clk after 25 ns;

end CntrTB_a;

```

**Figure 9.9.9-2 Counter Testbench (Potpouri\cntrtb.vhd)**

```

configuration CounterA_Cfg of CntrTB is
    for CntrTB_a
        for U1_Counter: Counter use
            entity Work.Counter(Counter_a);
            end for;
    end for;
end CounterA_Cfg;

configuration CounterRTL_Cfg of CntrTB is
    for CntrTB_a
        for U1_Counter: Counter use
            entity Work.Counter(Counter_Rtl);
            end for;
    end for;
end CounterRTL_Cfg;

```

Two configurations for the binding of the counter entity to one of its architectures.

**Figure 9.9.9-3 Counter Configuration (Potpouri\cntrcfg.vhd)**

### 9.9.10 Concurrent Statements

**Q** | What are the seven concurrent statements? Identify for each concurrent statement its advantages and typical applications.

**A** | Table 9.9.10 provides a summary of the concurrent statements and their attributes.

**Table 9.9.10 Concurrent Statements**

#	Concurrent Statement	Advantages	Typical Use
1	Process	Allows declaration of variables, and sequential statements including looping constructs	<ol style="list-style-type: none"> <li>1. Register inference with clocked statements</li> <li>2. Logic that operates on bits of a vector</li> </ol>
2	Concurrent signal assignment	Fast notation to imply combinatorial logic	Combinatorial logic
3	Component instantiation	Allows design hierarchy	<ol style="list-style-type: none"> <li>1. Hierarchical designs</li> <li>2. Testbench designs</li> </ol>
4	Block	Allows grouping of concurrent statements and localization of signals pertinent to a function.	<ol style="list-style-type: none"> <li>1. Synthesis for functional groupings</li> <li>2. RTL and testbenches for enhanced readability.</li> </ol>
5	Concurrent procedure call	Enables instantiation of functional elements without the overhead associated with components	In testbenches for monitoring of specified conditions or operations
6	Generate	Builds elements as specified in parameters.	<ol style="list-style-type: none"> <li>1. Builds the necessary number of elements in a design</li> <li>2. Enables the elaboration of specific monitoring or test functions in testbenches</li> </ol>
7	Concurrent assertion	Provides quick reporting of user defined violations	Error reporting of signal violations

### 9.9.11 Drivers and Resolution Functions

**Q** | When is a driver implied? When is it created? If a signal is of type Boolean, how many drivers are allowed on that signal? Define a package that provides a resolution function for the type Boolean where *false* overrides *true*. Define a resolved subtype named *rBoolean*. Write a model that uses that package.

**A** | Drivers are implied whenever a signal assignment is defined. If a process has at least one signal assignment statement, then a single driver (one and only one driver) is automatically created for that signal. If a process has more than one signal assignment for the same signal, there still is only ONE driver created for that signal. Signals defined

in an architecture do not have drivers because drivers are not associated with signal declarations. They are associated with signal assignments. Each signal assignment is said to be associated with a driver. Drivers are created during the elaboration of a design.

If a signal is of type Boolean, only one driver is allowed on that signal because type Boolean is an unresolved type. Figure 9.9.11 defines a package that provides a resolution function for the type Boolean where *false* overrides *true*. That figure also provides a model that makes use of the resolved Boolean type described in the package.

```

package Boolean_Pkg is
    type Bool_Array is array(Integer range <>) of Boolean;
    -- False wins over True
    function rBoolean(DRIVERS : Bool_Array) return Boolean;

    subtype rBoolean_Typ is rBoolean Boolean;
end Boolean_Pkg;

package body Boolean_Pkg is
    -- False wins over True
    function rBoolean(DRIVERS : Bool_Array) return Boolean is
        variable Found_False : Boolean := False;
    begin
        LoopThruAllDrivers: for Idx_i in DRIVERS'range loop
            if not DRIVERS(Idx_i) then -- there is a false
                Found_False := True;
                exit LoopThruAllDrivers;
            end if;
        end loop LoopThruAllDrivers;
        return not Found_False;
    end rBoolean;
end Boolean_Pkg;

entity TBoolean is
end TBoolean;

library Work;
use Work.Boolean_Pkg.all;
architecture TBoolean_a of TBoolean is
    signal A_s : Boolean;
    signal B_s : Boolean;
    signal C_s : rBoolean_Typ; ← C_s is a resolved signal

begin -- TBoolean_a
    T_Lbl: process -- generation of test signals
    begin -- process T_Lbl
        Hi_Lbl: for I in Boolean loop
            A_s <= I;
            wait for 10 ns;
        Lo_Lbl: for J in Boolean loop
            B_s <= J;
            wait for 10 ns;
        end loop Lo_Lbl;
    end loop Hi_Lbl;
    end process T_Lbl;

    C_s <= A_s; -- Concurrent statement ← One driver on C_s
    C_s <= B_s; -- Concurrent statement ← Another driver on C_s
end TBoolean_a ;

```

Simulation Results				
ns	delta	a_s	b_s	c_s
0	+0	false	false	false
20	+1	false	true	false
30	+2	true	true	true
40	+2	true	false	false

Figure 9.9.11 Resolved Boolean Function and Example (potpouri\rbool.vhd)

### 9.9.12 Subprogram

**Q**

Write a function that counts the number of ones in a Std\_Logic\_Vector, starting from the left until either an 'X' is encountered, or until the end of the vector.

**A**

Figure 9.9.12 provides a definition and an application of that function.

```

library IEEE;
use IEEE.Std_Logic_1164.all;
entity SubPrgm is
end SubPrgm;

architecture SubPrgm_a of SubPrgm is
function CountOnes(Value : Std_Logic_Vector) return Integer is
variable Data_v : Std_Logic_Vector(1 to Value'length);
variable Count_v : Integer;
begin
Data_v := Value;
Count_v := 0;
Count_Lbl : for I in Data_v'range loop
  if Data_v(I) = '1' then
    Count_v := Count_v + 1;
  elsif Data_v(I) = 'X' then
    exit Count_Lbl;
  end if;
end loop Count_Lbl;

return Count_v;
end CountOnes;
signal UP_s : Std_Logic_Vector(0 to 7) := "1110X001";
signal DOWN_s : Std_Logic_Vector(7 downto 0) := "1110X001";
signal IntUp_s : Integer;
signal IntDn_s : Integer;

begin -- SubPrgm_a
T_Lbl : process
begin -- process T_Lbl
  UP_s <= UP_s(1 to 7) & '1'; -- shift left
  DOWN_s <= DOWN_s(6 downto 0) & '1';
  wait for 20 ns;
end process T_Lbl;

IntUp_s <= CountOnes(UP_s);
IntDn_s <= CountOnes(DOWN_s);

end SubPrgm_a;
-- simulation results
--   ns  delta  up_s  down_s  intup_s  intdn_s
--   0   +2  110X0011 110X0011      2        2
--   20  +2  10X00111 10X00111     1        1
--   40  +2  0X001111 0X001111     0        0
--   60  +1  X0011111 X0011111     0        0
--   80  +2  00111111 00111111     6        6
--  100  +2  01111111 01111111     7        7
--  120  +2  11111111 11111111     8        8

```

Normalization of formal parameter

Initialization performed using a synthesizable approach (i.e., through an assignment statement rather than at variable declaration).

Figure 9.9.12 Subprogram Example (potpourri\subprgm.vhd)

## **10. DESIGN FOR REUSE**

---

---

In this context, design for reuse is the ability to define new architectures that are impermeable to new technologies and are malleable to new requirements. Design for reuse requires disciplines in design processes, VHDL coding styles and documentation. The disciplines in design processes address the issues of integrating pre-designed models and of entering into the reuse data base newly defined items. The VHDL Coding style discipline addresses the issues of partitioning, parameterization, and model design for readability, flexibility and reuse. The documentation discipline addresses the issues of good documentation to enable design reuse and better communication between the engineers who have specified the system and those who have implemented it or are applying it.

## 10.1 DESIGN PROCESSES FOR REUSABILITY

**Q** | What are the design processes that support design reuse and design flexibility to adapt to the modeling requirements?

**A** | Figure 10.1 demonstrates a generic design process that emphasizes design reuse. This process starts with a set of requirements that define what the implemented design must do, but not how it is implemented. For example, the requirements can define the necessity for a microwave controller that interfaces with a keyboard, a display, a temperature sensor, humidity sensor, and a microwave power controller. The requirements document must also include the algorithms for the controls.

Hardware represents a translation of those requirements. The general approach is a top-down and bottom-up approach. The requirements can be broken down into **functional partitioning** (top-down), taking into account core designs (bottom-up). For example, the functional partitioning may consist of a keyboard interface, a display interface, a sensor interface, a power controller interface, a controller, and a memory.

The functional partitions can make use of core designs provided in a library of available designs. Those designs may be available at either the RTL HDL format, or synthesized to a gate level format or netlist such as EDIF format. For example, a main controller can consist of a programmable micro-controller described in the core library. A discrete controller design can instead use discrete counters (from the core) along with a discrete state machine. A RAM model or a pre-designed display interface (from another design) described as a core can be reused. See section 10.2.9 for an expanded discussion of core designs.

Once the functional partitions are defined, each functional block can be further decomposed into its **design partitioning**, taking into account the data flow and control flow of its elements. This design stage characterizes the data registers, data processing (e.g., ALUs, multiplexers), and the controls necessary for the partition architecture. In some cases, a **timing interface definition and analysis** is necessary to characterize and document the timing interfaces of the design. Figure 10.3.5-1 demonstrates an example of a design partitioning with the data flow and control flow.

Following the design partitioning stage, the **VHDL RTL modeling** can proceed with the scalability and design for reuse guidelines defined in section 10.3. The design can then proceed to **functional verification** through simulation using a testbench. The testbench design (see chapter 8) may reuse core designs (e.g., components, packages) previously defined and available for use. Examples of core packages include models (such as the memory model, chapter 6) and packages (such LFSR, MISR, Image, chapter 5). **Synthesis** of the RTL design provides detailed design information targeted toward a specified technology using vendor's libraries. **Synthesis, routing, and pre and post layout timing analysis** provide information necessary to tune the RTL/synthesis processes in order to meet the timing requirements.

The design can be fabricated once it is verified, routed, and analyzed for timing. Documentation of the design and its processes is necessary to enhance the reusability of the design. The reusable components (e.g., hardware models, packages) can then be added to the core design library.

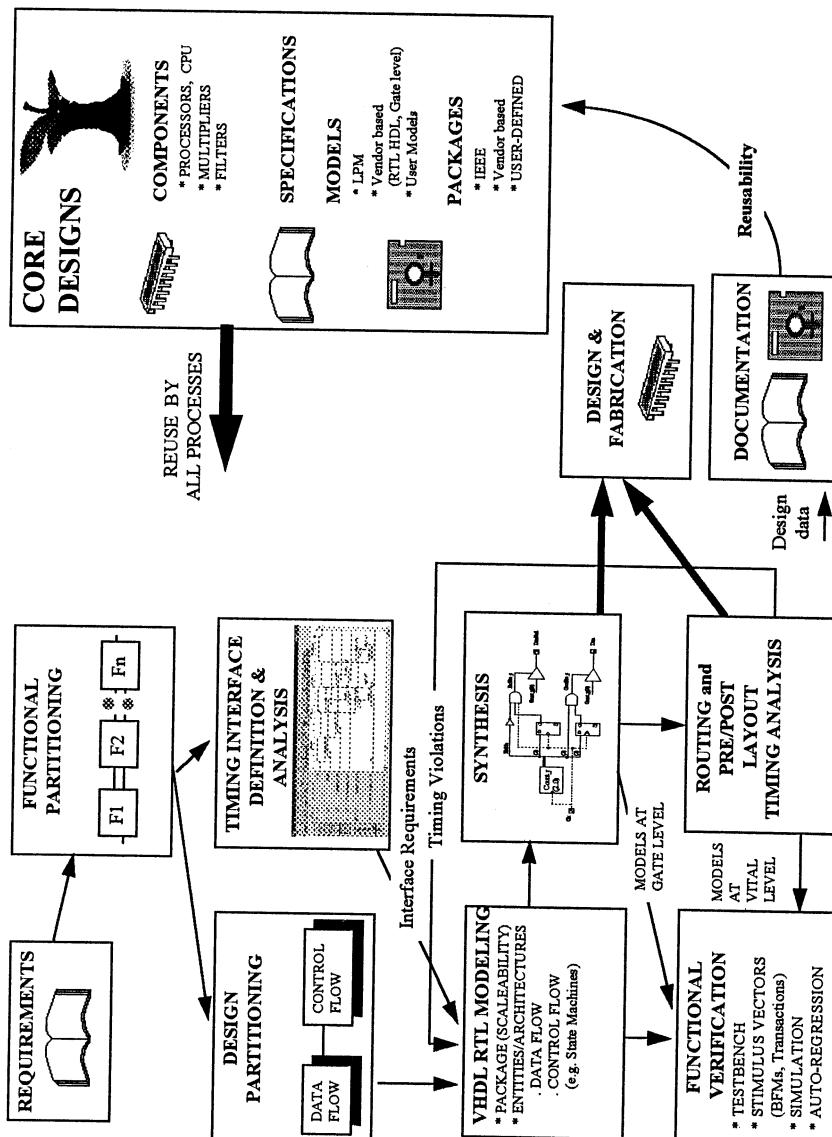


Figure 10.1 Design Processes for Reusability

## 10.2 PARAMETERIZED, REUSABLE AND READABLE CODE

**Q** What methods can be used to create parameterized, reusable, and readable code? Specifically, how can bus widths, address decoding values, and location of control fields be specified such that they can easily be modified?

**A**  It is generally recommended to avoid hard coded values because of the difficulty in making changes. Parameterization provides the flexibility in defining a model's malleability. Parameterization is addressed by many coding style disciplines depending upon the classes of problems. There are two classes of parameters:

- **"Accordion" parameters.** These parameters specify variations in the model's object widths (e.g., signal widths), or depths (e.g., register depths), or timing (e.g., delay parameters). The term "accordion" is coined to exemplify the linear variations in the values of the parameters that expand or contract as a function of the design requirements.
- **"Functional" or "Control" parameters.** These parameters specify functional information and variations in the model's address decoding values, and location and values of control fields that define specific operations. These types of parameters are common in the design of controllers with microprogramming like control fields. In a typical design process, the location and values of these fields often change until the design specifications are frozen.

Design reusability and readability are addressed not only through parameterization, but also through the use of coding styles. This section addresses those issues.

### 10.2.1 Notation

A design contains several design units and objects of various types that belong to classes (i.e., *constant*, *signal*, *variable*, *file*). Identifying these items with an easily recognizable notation enhances readability. The suffix notation defined in table 10.2.1 was found very helpful. It represents a variation to the Hungarian notation often used in C++ applications. In addition, many synthesizers or timing analysis tools provide information about implied registers. The use of this notation facilitates the user's recognition of those registers. For example, if a signal with an *\_s* or *\_v* suffix notation is implied by the synthesizer as a register, then the possibility of an error may exist.

*COMMENTS:* For simple loops, users have experienced the characters 'I', 'J', 'K' as more practical for loop indices. However, for more complex loops, where additional meaning is attached to the loop index, the use of the *\_i* suffix is recommended. In addition, the differentiation between locally static versus a globally static (see section 10.2.3) is maintained in the notation for constants only (with *\_c* or *\_gc* suffixes). That differentiation is not carried to types, signals, or variables because it was felt to be excessively burdensome. A user who wants to keep track of the globally static objects may however carry that notation further with *\_gs*, *\_gr*, *\_gv*, *\_gvc*, or *\_gtyp*.

Table 10.2.1 Recommended Suffix Notation

Suffix	Explanation	Example
<u>Pkg</u>	Package identifier	<code>package Design_Pkg is</code>
<u>Lib</u>	Library identifier	<code>Library ProjectX_Lib;</code>
<u>Cfg</u>	Configuration identifier	<code>configuration EEPROM_Cfg of</code>
<u>Typ</u>	Type or subtype identifier	<code>subtype Read_Typ is Integer range 20 downto 18</code>
<u>a,</u> <u>Beh,</u> <u>Fnc</u> <u>Rtl,</u> <u>Str,</u>	architecture (general case) Behavior level Functional (e.g. ISA level) Register transfer level Structural level	<code>architecture EntityName_a of EntityName is</code> <code>architecture EntityName_Beh of EntityName is</code> <code>architecture EntityName_Fnc of EntityName is</code> <code>architecture EntityName_Rtl of EntityName is</code> <code>architecture EntityName_Str of EntityName is</code>
<u>Blk</u>	Block label	<code>ALU_Blk: Block</code>
<u>Lbl</u>	Label for non-blocks	<code>Alu_Lbl: process</code>
<u>v</u>	Variable implemented as combinatorial logic	<code>variable Data_v : Std_logic_Vector(31 downto 0)</code>
<u>vr</u>	Variable implemented as register	<code>variable Dat_vr : Std_logic_Vector(Read_Typ)</code>
<u>s</u>	Signal implemented as a bus	<code>signal Data_s : Std_logic_Vector(31 downto 0)</code>
<u>r</u>	Signal implemented as register	<code>signal Data_r : Std_logic_Vector(31 downto 0)</code>
<u>c</u>	Constant, locally static	<code>constant Read_c : Std_Logic_Vector(Read_Typ) := "001";</code>
<u>gc</u>	Constant, globally static	<code>constant Size_gc := 2 ** NumBits_g;</code>
<u>g</u>	Generic identifier	<code>generic(DataWidth_g : Integer := 32);</code>
<u>I</u>	Index for a loop	<code>for Bit_i in Count_v'range loop</code>
<u>f</u>	File identifier	<code>file Data_f : Std.TextIO.Text is</code>

## 10.2.2 Path Definition

It is generally considered good software engineering practice to include the package path when declaring or accessing those objects. This is particularly true when accessing global signals or variables because it enhances comprehension (e.g., `Work.My_Pkg.GobalSignal <= SomeValue`). However, the inclusion of the path for every object declared in packages can be burdening and may lessen readability. In addition, synthesis may not support this selected name coding style. The general recommendation for non-synthesizable designs is to include the path unless the following conditions are true:

1. The objects are well understood (such as `Std_Logic` type);
2. There are few packages in the design, and it is clear from the context where the objects are defined. In those cases, a few comment lines can clearly help the reader as to the package being used.

Below is an example of a complete path definition where the integer subtype `Read_Typ` is defined in package `Design_pkg`.

`signal Read_s : Std_Logic_Vector(Work.Design_Pkg.Read_Typ);` 

Many designers object to the long names and feel that this approach clutters readability. An optional approach is to bypass the library name as shown in figure 10.2.2.

```

library Work;
use Work.Design_Pkg.all;      ← This is necessary when accessing the implied
                               operators for enumerated data types
use Work.Design_Pkg;          ← This bypasses the need to use the library name
...
architecture Design_a of Design is   ← Path name (without library name)
  signal Read_s : Std_Logic_Vector(Design_Pkg.Read_Typ);
begin
  Design_Pkg.GlobSig_s <= Read_s; -- assignment to a global signal

```

Figure 10.2.2 Technique to bypass the Library Name in Path Definition

### 10.2.3 “Accordion” Parameterization

The accordion parameters can be described either with *generics* or with *constants*. *Generic* parameters and *deferred constants* are *globally static*. It is important to understand the difference between a *locally static* and a *globally static* expression and the restrictions of *globally static* expressions. [1]A *locally static expression* is an expression that can be evaluated during the *analysis of the design unit in which it appears* (e.g., a constant initialized with a locally static expression). A *globally static expression* is an expression that can be evaluated *as soon as the design hierarchy in which it appears is elaborated*. Section 7.4.1 and 7.4.2 of the LRM defines all the cases that qualify locally static and globally static primaries. Table 10.2.3 provides examples of *globally static* expressions.

Table 10.2.3 *Globally Static Expressions*

EXPRESSION	COMMENT
generic(Size_g : Integer := 6);	Value of <i>Size_g</i> is determined at elaboration.
constant Size_gc : Integer := 2**Size_g;	Value of <i>Size_gc</i> is dependent on the globally static generic <i>Size_g</i> .
package T_Pkg is constant Bits_gc : Integer; end T_Pkg;	<i>Bits_gc</i> is a <i>deferred</i> constant, and is globally static. A recompilation of <i>T_Pkg body</i> does not require a recompilation of an architecture that uses the package.
constant Sync_c : Time := 300 ns;	A literal of type <i>Time</i> is globally static.
subtype ABC_Typ is Std_Logic_Vector (Size_g - 1 downto 0);	Range of subtype <i>ABC_Typ</i> is globally static. Objects of that subtype are globally static
signal ABC_s Std_Logic_Vector (Size_g - 1 downto 0) := (others => '0');	Range of signal <i>ABC_s</i> is globally static. Subtype of <i>ABC_s</i> is globally static
case ABC_s'length is -- ●*	Attribute is a function of a globally static expression. Case expressions must be locally static,

Since *generics* are evaluated at elaboration (rather than compilation) they are ideal in providing modeling information that is characteristic for each component instantiation. For example, component delay parameters, or memory depth size that may be different for each component instantiation of a memory model can be tuned using generics. Configurations can then be used to declare different values for the generics in each of declarations. However, *generics* are globally static parameters and include design restrictions. For example, *generics* cannot be used in *case* expressions or *case* choices. However, locally static *constants* (declared in a package or in an architecture) are *locally static*, and can be used in *case* expressions or *case* choices. Figure 10.2.3 demonstrates the limitations of *globally static* objects.

```
entity Global is
  generic (Size_g : Integer := 8);
  port (DataIn : in Bit_Vector(Size_g - 1 downto 0);
        DataOut : out Bit_Vector(Size_g - 1 downto 0));
end Global;
architecture Global_a of Global is
begin -- Global_a
  Test_Lbl : process (DataIn)
    variable Size_v : Integer;

  begin -- process Test_Lbl
    if DataIn = "00000000" then
      DataOut <= "10100000";
    end if;
    case DataIn is
      when "00000000" =>
        DataOut <= "10100001";
      when others =>
        null;
    end case;

    case Size_v is
      when Size_g => Size_v := 0;
      when others => null;
    end case;
  end process Test_Lbl;
end Global_a;
```

 Array case expression must have a static subtype.

 Case choice must be a locally static expression.

**Figure 10.2.3 Limitations of Globally Static Parameters (\reuse\global.vhd)**

Section 7.12 provides an example of the use of *generics* to parameterize a design. Section 7.13 provides an example of the use of *constants* in a *package* to parameterize a priority encoder.

#### 10.2.4 “Control” Parameterization

In the design of control machines, it is common practice to have various control fields provide control information. These fields are typically sourced from registers or ports. They provide microprogramming like information such as source selects, or micro-operations, or output enables. Since these control fields may vary in size and location, it is recommended to parameterize these fields. The following options are available:

1. **All** VHDL generics can be used to parameterize bus widths, start and end locations of control fields, or design options. However, because of the globally static limitations associated with *generics* (see above), they are not recommended for this purpose.
2. **All** Use of integer subtypes based on locally static constants is recommended to parameterize ranges for control fields. Use of locally static constants is also recommended to define data fields such as address decode values. When subtypes are used to define vector slices (e.g., *BusA(Src\_Typ)*), they not only provide flexibility in redefining the ranges, but their names enhance code readability because they qualify the associated functions. Those constants and slices are locally static, and thus can be used in *case* expressions. Defining these subtypes and constants in packages rather than in the architecture of designs is recommended because other designs (such as variations to the original architecture and testbenches) would automatically be bound to those packages. This eliminates the need for multiple copies of the same definitions, and minimizes issues related to configuration management of those parameters. **Packages provide a single definition source for the defined parameters that are bound by the compilation process.** A change to a package declaration requires a recompilation of the design units that make use of that package.
3. **All** The definition of aliases can also be used to parameterize control fields. However, aliases are not synthesizable. The use of aliases inhibits the direct reuse of the control definitions.
4. **All** Records can provide an alternate solution to identify fields of a control signal. They enable the definition of different types for each field (e.g., *Boolean* for a single bit control field). However, conversion functions are required to translate between the signal type (e.g., *Std\_Logic\_Vector*) and the *record* type. In addition, some synthesizers do not accept *records*.

The following guidelines are recommended: **All**

In a user defined package:

1. **Define subtypes for each of the control fields using integer ranges.**

```
Subtype Source_Typ is Integer range 20 downto 18;  ?  
Subtype Addr_Typ is Integer range 7 downto 0;
```

Warning: The use of types or subtypes as aggregate choices may not be supported by synthesis. Some synthesizers support the use of subtypes as aggregate choices when they describe discrete ranges (e.g., *SomeSignal(Source\_Typ)*). Synopsys® compiler is fully compliant.

As an alternate solution, declare locally static constants that delimit the high and low values of each of the control fields.

```
constant SourceHi_c : Integer := 20;  
constant SourceLo_c : Integer := 18;  
...
```

Note: A signal range can be defined as follows:

*SomeSignal(SourceHi\_c downto SourceLo\_c)*

2. **Define locally static constants to represent values for control fields and address decoding values.**

```
constant SourceA_c : Std_Logic_Vector(Source_Typ) := "000";
constant RegFile_c : Std_Logic_Vector(Source_Typ) := "011";
Constant AddrRegFile_c : Std_Logic_Vector(Addr_typ) := "00000010";
...
```

3. Alternatively, **define enumerated data types to represent states of control fields.** Use conversion functions to translate between bit field vectors and enumerated types.

```
type Opr_Typ is (Pass, Add, Sub, Fand);
function ToOpr(S : Std_Logic_Vector) return Opr_Typ;
function ToStd(S : Opr_Typ) return Std_Logic_Vector;
```

In the architecture:

4. **Use the defined subtypes to access fields of the control registers,** e.g.,

```
case ControlReg(Source_Typ) is . . .
for I in Source_Typ loop      -- 20 downto 18
  B <= (B'high - 1 downto Source_Typ'high + 1 => '0',   -- 32 .. 21
         Source_Typ                      => '1',   -- 20..18
         others                          => '0'); -- 17 .. 0
BusA(Source_Typ) <= RegFile_c; -- constant defined in package
```

5. **Use the defined constants to access the range definition.**

```
Case ControlReg(SourceHi_c downto SourceLo_c) is
```

6. **Use conversion functions to translate bit fields to enumerated data types.** e.g.,

```
Case ToOpr(ControlReg(Source_Typ)) is
  when Pass => . . .
```

7. **Use constants specified in the user packages.**

Constants specify values that signals or variables can either take or must be compared with, e.g.,

```
when SourceA_c => ...
when RegFile_c => ...
if BusA(Addr_Typ) = Status_c then . . .
```

Figure 10.2.4 demonstrates the concepts of defining and using parameters for design parameterization.

```

package My_Pkg is
    Subtype Source_Typ is Integer range 20 downto 18;
    Subtype Cmd_Typ is Integer range 1 downto 0;
    Subtype Addr_Typ is Integer range 7 downto 0;
    type Opr_Typ is (Pass, Add, Sub, Fand);

    constant SomeAddr_c : Bit_Vector(Addr_Typ) := "10101010";

    function ToOpr(S : Bit_Vector(1 downto 0)) return Opr_Typ;
end My_Pkg;

package body My_Pkg is
    function ToOpr(S : Bit_Vector(1 downto 0)) return Opr_Typ is
        begin
            case S is
                when "00"      => return Pass;
                when "01"      => return Add;
                when "10"      => return Sub;
                when others   => return Fand;
            end case;
        end ToOpr;
end My_Pkg;

library Work;
use Work.My_Pkg.all;
entity T is
    port (A    : in     Bit_Vector(31 downto 0);
          B    : out    Bit_Vector(31 downto 0));
end T;

architecture T_a of T is
begin -- T_a
    Test_Lbl : process(A)
    begin -- process Test_Lbl
        case A(Source_Typ) is
            when "000" =>
                for I in Source_Typ loop      -- 20 downto 18
                    if I = Source_Typ'high then
                        B <= (B'high - 1 downto Source_Typ'high + 1 => '0',  -- 32 .. 21
                                Source_Typ           => '1',  -- 20..18
                                others                 => '0'); -- 17 .. 0
                    end if;
                end loop;

            when others =>
                B(Addr_Typ) <= SomeAddr_c;
        end case;

        case ToOpr(A(Cmd_Typ)) is
            when Pass  => B <= A;
            when Add   => B <= not A;
            when Sub   => null;
            when Fand  => B <= (Addr_Typ => '1',
                                  others    => '0');
        end case;
    end process Test_Lbl;
end T_a;

```

**Subtypes as array range definition  
enhance reusability**

**Subtypes as loop indices or aggregate  
choices enhance reusability**

**Type conversion for enhanced readability  
of case choices**

Figure 10.2.4 Defining and Using Parameters for Design Parameterization  
(reuse\aggr.vhd)

### 10.2.5 *Block* Statements for Partition Separations

*Blocks* provide the following advantages in functional partitions:

- **Enhanced readability.** Each *block* identifies a separate functional partition in an architecture. Components provide a stricter functional separation than *blocks* because the visibility of the interfaces is limited to the ports of the component (see section 4.1). However, unlike components, *blocks* have visibility of all the architectural top level declarations (i.e., architectural signals, types, constants, subprograms). Thus *blocks* provide a hybrid between components (because of the local block declarative items) and a flat design (without the blocks).
- **Information hiding.** Busses, registers, functions and subprograms local to a partition can be defined within the block declarative section of the *block*. Ports or guards in blocks should not be used because they are generally not synthesizable. When a signal local to a block must be visible to the top level architecture, the following guidelines are recommended:
  - \* Define the signal (that represents a bus or a register) inside the block declarative part. This causes that bus or register to be local to block, and to belong to the partition that the block represents.
  - \* Within the *block*, use a concurrent signal assignment to transfer the value of the desired local signal onto an architectural signal (declared in the declarative part of the architecture) or port of the entity.

- **Easy grouping and faster synthesis.** With Synopsys® synthesizer, use the following compile directives to group all HDL created blocks into their own level of hierarchy: *group -hdl\_all\_blocks*

This compilation directive not only allows for faster synthesis, but also enables the separate viewing of each block (for enhanced understanding) with *design analyzer* (a tool with a mode to view the compiled designs).

Figure 10.2.5-1 defines an architectural block diagram demonstrating the use of *blocks*. Figure 10.2.5-2 represents the VHDL code for that architecture.

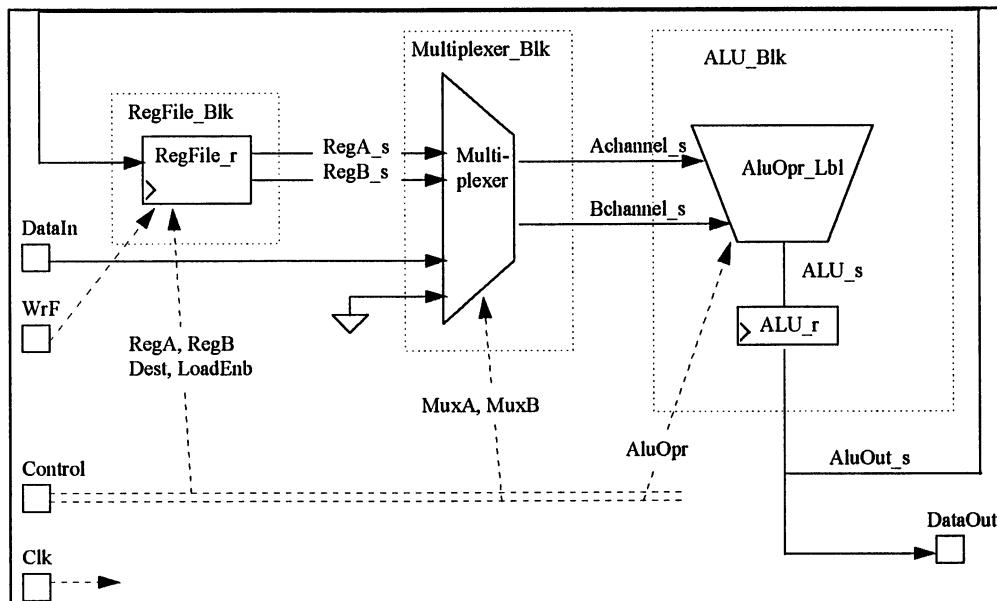


Figure 10.2.5-1 Architectural Diagram Demonstrating Use of *Blocks*

```

library IEEE;
use IEEE.Std_Logic_1164.all;

package Def_Pkg is
    subtype Bits32_Typ is Integer range 31 downto 0;
    constant RegFileAddr_c : Integer := 3; -- # of address bits
    type RegFile_Typ is array(RegFileAddr_c - 1 downto 0)
        of Std_Logic_Vector(Bits32_Typ);

    subtype RegA_Typ is Integer range 31 downto 31 - RegFileAddr_c; -- 31.29
    subtype RegB_Typ is Integer range 27 downto 27 - RegFileAddr_c; -- 27.25
    subtype MuxA_Typ is Integer range 23 downto 22;
    subtype MuxB_Typ is Integer range 21 downto 20;
    subtype AluOpr_Typ is Integer range 15 downto 13;
    subtype Dest_Typ is Integer range 10 downto 10 - RegFileAddr_c; -- 10.8
    subtype LoadEnb_Typ is Integer range 0 downto 0; -- enable load
    -- MuxA selects RegA
    constant MuxA_A_c : Std_Logic_Vector(MuxA_Typ) := "00";

    -- MuxA selects RegB
    constant MuxA_B_c : Std_Logic_Vector(MuxA_Typ) := "01";

    -- MuxA selects DataIn
    constant MuxA_D_c : Std_Logic_Vector(MuxA_Typ) := "10";

    -- MuxA selects Zero
    constant MuxA_0_c : Std_Logic_Vector(MuxA_Typ) := "11";

    -- MuxB selects RegA
    constant MuxB_A_c : Std_Logic_Vector(MuxB_Typ) := "00";

    -- MuxB selects RegB
    constant MuxB_B_c : Std_Logic_Vector(MuxB_Typ) := "01";

    -- MuxB selects DataIn
    constant MuxB_D_c : Std_Logic_Vector(MuxB_Typ) := "10";

```

```

-- MuxB selects Zero
constant MuxB_0_c : Std_Logic_Vector(MuxB_Typ) := "11";

-- ALU Operations
constant Plus_c    : Std_Logic_Vector(AluOpr_Typ) := "000";
constant Min_c     : Std_Logic_Vector(AluOpr_Typ) := "001";
-- . .
constant And_c     : Std_Logic_Vector(AluOpr_Typ) := "110";

constant PassA_c   : Std_Logic_Vector(AluOpr_Typ) := "100";
constant PassB_c   : Std_Logic_Vector(AluOpr_Typ) := "101";
constant XOR_c     : Std_Logic_Vector(AluOpr_Typ) := "111";
-- . .
constant Zero32_c  : Std_Logic_Vector(Bits32_Typ) := (others => '0');
end Def_Pkg;

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_Arith.all;

library Work;
use Work.Def_Pkg.all; -- includes subtypes and definitions of constant

entity T is
  -- Clk      : Clock
  -- DataIn   : Input data
  -- Control  : Control of T. Fields for the control are
  --            : defined in the Def_Pkg package and include:
  -- WrF      : Write enable too load register file
  -- DataOut   : Output data
  port (Clk      : in      Std_Logic;
        DataIn   : in      Std_Logic_Vector(Bits32_Typ);
        Control  : in      Std_Logic_Vector(Bits32_Typ);
        WrF      : in      Std_Logic;
        DataOut   : out     Std_Logic_Vector(Bits32_Typ));
end T;

architecture T_a of T is
  -- RegFile outputs
  signal RegA_s : Std_Logic_Vector(Bits32_Typ);
  signal RegB_s : Std_Logic_Vector(Bits32_Typ);

  -- Multiplexer Outputs
  signal Achannel_s : Std_Logic_Vector(Bits32_Typ);
  signal Bchannel_s : Std_Logic_Vector(Bits32_Typ);

  -- Alu output
  signal AluOut_s : Std_Logic_Vector(Bits32_Typ);

begin -- T_a
  -----
  -- DataOut assignment, at top level
  -----
  DataOut <= AluOut_s;           ← AluOut_s signal can be read by other
                                concurrent statements.

```

Path not defined  
because only one user  
define package is used.

```

-- Register File Block
-- Defines reading and writing of data from/into register file
-----  

RegFile_Blk: block
    -- register file
    signal RegFile_r : RegFile_Typ;
-----  

begin -- block RegFile_Blk
-----  

    -- Process: RegA_Lbl
    -- Purpose: Reading of register file
-----  

    RegA_Lbl : process(Control(RegA_Typ), Control(RegB_Typ))
        variable IA_v : Integer;
        variable IB_v : Integer;
    begin -- process RegA_Lbl
        IA_v := CONV_INTEGER(Unsigned(Control(RegA_Typ)));
        IB_v := CONV_INTEGER(Unsigned(Control(RegB_Typ)));
        RegA_s <= RegFile_r(IA_v);
        RegB_s <= RegFile_r(IB_v);
    end process RegA_Lbl;
-----  

    -- Process: LoadRegFile_Lbl
    -- Purpose: Load register file, as controlled by the control input
    --           (LoadEnable) and the WrF input
-----  

    LoadRegFile_Lbl : process
    begin -- process LoadRegFile_Lbl
        wait until Clk'event and Clk = '1';
        if Control(LoadEnb_Typ) = "1" and
            WrF = '0' then
            RegFile_r(CONV_INTEGER
                (Unsigned(Control(Dest_Typ)))) <= AluOut_s;
        end if;
    end process LoadRegFile_Lbl;
end block RegFile_Blk;
-----  

-- Multiplexer Block
-----  

Multiplexer_Blk: block
begin -- block Multiplexer_Blk
-----  

    -- Concurrent signal assignment Style
    Achannel_s <=
        RegA_s when Control(MuxA_Typ) = MuxA_A_c else
        RegB_s when Control(MuxA_Typ) = MuxA_B_c else
        DataIn when Control(MuxA_Typ) = MuxA_D_c else
        Zero32_c; -- MuxA_0_c
-----  

    -- process style
-----  

    -- Process: Bchannel_Lbl
-----  

    Bchannel_Lbl : process (RegA_s, RegB_s, DataIn, Control(MuxB_Typ))
    begin -- process Bchannel_Lbl
        case Control(MuxB_Typ) is
            when MuxB_A_c =>
                Bchannel_s <= RegA_s;
            when MuxB_B_c =>
                Bchannel_s <= RegB_s;
            when MuxB_D_c =>
                Bchannel_s <= DataIn;
        end case;
    end process Bchannel_Lbl;
-----  


```

**Register file block with 2 processes:**

- One for READ
- One for WRITE

**Multiplexer block with 2 concurrent statements:**

- A Channel
- B Channel

```

when MuxB_0_c =>
  Bchannel_s <= Zero32_c;

when others =>
  null;

end case;
end process Bchannel_Lbl;
end block Multiplexer_Blk;

-----
-- ALUBlock
-----
Alu_Blk: block
  signal ALU_r : Std_Logic_Vector(Bits32_Typ);
  signal ALU_s : Std_Logic_Vector(Bits32_Typ);

begin -- block ALU_Blk
  -- Process:   ALU_Lbl
  -- Purpose:   ALU Operation

  AluOpr_Lbl : process(Achannel_s, Bchannel_s, Control(AluOpr_Typ))
  begin -- process AluOpr_Lbl
    case Control(AluOpr_Typ) is
      when Plus_c  =>
        Alu_s <= Unsigned(Achannel_s) + Unsigned(Bchannel_s);

      when Min_c   =>
        Alu_s <= Unsigned(Achannel_s) - Unsigned(Bchannel_s);

      when And_c   =>
        Alu_s <= Achannel_s and Bchannel_s;

      when PassA_c =>
        Alu_s <= Achannel_s;

      when PassB_c =>
        Alu_s <= Bchannel_s;

      when Xor_c   =>
        Alu_s <= Achannel_s xor Bchannel_s;

      when others   =>
        null;
    end case;
  end process AluOpr_Lbl;

  -- Process:   AluReg_Lbl
  -- Purpose:   Clocks ALU_s into register ALU_r
  -----
  AluReg_Lbl : process
  begin -- process AluReg_Lbl
    wait until Clk'event and Clk = '1';
    ALU_r <= ALU_s;
  end process AluReg_Lbl;

  -----
  -- Transfer local register to top-level signal
  -----
  AluOut_s <= ALU_r;
end block Alu_Blk;
end T_a ;

```

**ALU Block with**

- Local Register (*ALU\_r*)
- Local signal (*ALU\_s*)

**Block includes 3 concurrent statements:**

- ALU operation process
- Local register loading process
- local register transfer to top-level signal

Figure 10.2.5-2 Use of Subtypes and constants (reuse\param3.vhd)

### 10.2.6 Generate Statement for Multiple Instantiations of Components

Again, the key to reusability is to avoid the hard wiring or instantiation of components. Instead, component instantiations can be parameterized through the use of generics or constants and through the *generate* statement to instantiate the required component instances. The *generate* statement can also be used to selectively enable a set of concurrent statements used for debug. Section 1.3.4 and 7.12 provide application examples of the *generate* statement.

### 10.2.7 Subprograms

 Subprograms enhance the concept of reusability because they capture the essence of the intended algorithms or functions. Subprograms are reusable because they can be called multiple times. Subprograms should NOT have side effects (see section 4.1). Subprograms that serve common sets of tasks should be included in packages. This enhances the cohesiveness of the design because they can be collectively grouped and reused. This compares to subprograms separately defined in architectures, with no direct reusability for other architectures (except through a copy and paste process).

 Subprograms should be written with unconstrained formal parameters to enable maximum reusability. In addition, subprograms must not rely on the direction and range of the actual parameters, and normalization is necessary (see section 4.8).

### 10.2.8 Use of VHDL Attributes

 Attributes provide a mechanism to enhance the flexibility and reusability of a design because they can extract information about design parameters. For example, a change to an object parameter (such as its size, range, or length) can be easily accessed with attributes without a coding change. Attributes can be used to specify discrete ranges without the use of discrete numbers. Thus, if the range of the parameter changes, the body of the model needs no modification. Examples include:

```
for I in S1'range loop -- loop range
for I in (S1'length - 1) downto 0 loop
for I in S'high to S'low -- loop range
  S(S1'range ) <= "1010"; -- slice
  S2(S1'range  => '1', -- aggregate
      S'range    => '0', -- choices
      others     => 'L');
```



The use of attributes  
enhance code reusability

In subprograms, it is good practice to define the formal parameters that are arrays as unconstrained (versus constrained) arrays. Attributes are then used to constraint objects local to subprograms (e.g., constants, variables, signals). See section 4.8 for a discussion and examples of attributes in subprograms.

### 10.2.9 Core Models

 The concept of reusable core models emerged from the development of medium scale integrated (MSI) and large scale integrated (LSI) components. These components included devices such as registers, register files, counters (up, down, up or down), multiplexers, arithmetic logic units (ALU), etc.). As the technology progressed, cascadable and slice devices were then offered as other sets of cores. These included components such as priority encoders, ALUs, multipliers, DMA slices, program control units (sequencer), I/O port devices, multiport pipeline processors, etc. For FPGA and ASIC designs, these concepts evolved to libraries of components such as Synopsys *DesignWare*.

In 1996, a new standard evolved for the definition of Library of parameterized models (*LPM*). The *LPM 2 1 0* standard was approved by the *EIA*. *LPM 2 1 0* is an adjunct to both the *EDIF 2 0 0* standard and the *EDIF 3 0 0* standard. It is now officially *EIA* Interim Standard #IS-103-A and is available for purchase from:

Electronic Industries Association                    (703) 907-7545 (voice)  
2500 Wilson Blvd.                                    (703) 907-7501 (FAX)  
Arlington, VA 22201                                <http://www.edif.org>

*Altera* has released to the public domain a set of components compliant with the *LPM 2 1 0* standard. This package is available at the following web site:

<http://www.edif.org/edif/lpmweb/>

Other vendors provide megafunction core models that can be directly used in designs. As of the date of this publication, these cores address many design disciplines including:

- Busses and interfaces IEEE 1284 parallel slave interface; PCI bus master and target; Power PC bus master, slave, arbiter; UART.
- Processors and peripherals Microprogram controller (2910, 49410); microprocessor (6502, Z80); microcontroller (8051).
- Telecommunication and data communications 6850 asynchronous communications interface adapter; HDLC data link controller; Reed-Solomon decoder; speed bridge FIFO.
- Digital Signal Processing Filter (adaptive, biorthogonal Wavelet, decimating); discrete cosine transform; FIR filter; floating point adder, divider, multiplier; image processing megafunctions (edge detection, image enhancement filters, video convolvers, etc.); JPEG decoder and encoder;

Many of these megafunction models are available in post-synthesis vendor specific format, in optimized netlists, and in VHDL and Verilog. A sample list of megafunction vendors can be found at the following URLs:

<http://www.altera.com/html/programs/alliances.html>.

[http://vhdl.org/vi/fmf/wwwpages/model\\_hotlist.html](http://vhdl.org/vi/fmf/wwwpages/model_hotlist.html)

This list is by no means exhaustive, and a user must conduct a search of companies that provide models. This list is summarized on disk in file *reuse\model\_ho.txt*.

The use of the core model concept enhances the reusability of designs. Section 7.13 provides an example of a priority encoder (or *first one* model) that takes advantage of a

reusable priority encoder core. That core is equivalent to the *Motorola MC14532B* 8-bit priority encoder.

### 10.2.10 Bus Functional Model (BFM) and Testbench Architecture

The key to reusability in the designs of BFM<sup>s</sup> is the **separation between the low-level protocols and the high-level sequencing of scenarios**. The low-level protocols represent the detailed twiddling of the interface signals. The high-level scenarios represent the high level transactions such as READ, WRITE, DMA. This separation provides the following advantages in the issues of reusability:

1. **Documentation and Readability.** A design is more apt to be reused when it can be understood and can be easily modifiable.
2. **Reuse of high-level scenarios.** A BFM design that requires a different bus protocol (or a change to some of the low-level protocols) may still reuse the high-level scenarios that specify the sequencing of the high-level transactions.
3. **Reuse of low-level protocols.** A BFM design that requires a change to the sequencing of the high level transactions can reuse the low-level protocols.

### 10.3 DOCUMENTATION OF VHDL DESIGNS

**Q** | What information should be included in the documentation of VHDL designs?

**A** | Good documentation is keyed to design **usability, maintenance, and reusability**. Even though VHDL is a hardware description language that is self-documenting, it is generally recommended to write an engineering design notebook to describe the design. This documentation is very useful for non-VHDL users who need to understand the design and its performance, as well as for VHDL users who must understand, maintain, and potentially upgrade and reuse the design. This notebook is also useful as a medium to describe the design to an audience, such as at design reviews. This notebook may consist of one or more documents depending upon the design complexity and the allotted labor. It can also be partitioned into the following documents:

- Users guide and data sheets
- Detailed design document

The following outline provides documentation guidelines for an engineering notebook.

#### 10.3.1 Summary

This section provides general information about the design, its performance (e.g., size, speed, number of gates, etc.), and its status including verification, synthesis, and physical design. It may provide major technical features of the design, such as its applicability to various systems. It may also include critical dates and schedules related to the design (e.g., design start, design reviews, design completed, design verified).

#### 10.3.2 Applicable Documents/References

This section references all applicable documents that pertain to the design including **design requirements, interface specifications, and general specifications**. Also referenced are any professional or industrial documents, such as the *IEEE Std 1076 VHDL Language Reference Manual*, *IEEE Std 1164 Multivalue Logic System for VHDL Model Interoperability*, and *IEEE P1076.3 Numeric packages*. Other references may include support documents and packages provided by synthesis and simulator vendors.

#### 10.3.3 Definitions

Abbreviations and terms used throughout the design description are defined in this subsection.

#### 10.3.4 Design Team

The design team who worked on the project must be identified to provide a reference of contact points in the event the design is updated or resurrected. Table 10.3.4 demonstrates a template for this description.

**Table 10.3.4 Design Team**

NAME	ENGINEERING CAPACITY	ORG	Phone	Address	Comments
---	(Project Manager, Design requirements, VHDL RTL code designer, synthesis translation, Testbench designer, Hardware design consultant, Synthesis consultant, etc.)	---	---	email, Mail	---
---	---	---	---	---	---

### 10.3.5 Requirement Versus Design Matrix

The design implementation must, by definition, meet the requirements imposed on the design. A “requirement versus design” matrix provides a summary of these requirements and demonstrates how the design meets those requirements. This information is useful in backtracking the rationales for the design decisions. Table 10.3.5 demonstrates a sample of such a matrix.

**Table 10.3.5 Sample Requirement Versus Design Matrix**

#	Requirement	Architecture
1	32-bit countdown timer with a programmable resolution ranging from 40 ns to 80 ns.	<ul style="list-style-type: none"> <li>• Architecture makes use of a 50 MHz internal clock.</li> <li>• Architecture makes use of a 12-bit countdown counter (<i>Counter_r</i>) clocked with the system clock. The value of this counter is reset to a value defined in control register (<i>Cntrl_r</i>). When the counter reaches a zero value, it is automatically reloaded to the reset value, thus providing a programmable resolution. The resolution can be scaled from 40 nanoseconds to 81.9 microseconds.</li> </ul>
--	--	• ---

### 10.3.6 Design Description

This section describes the design and includes, at a minimum, the following information.

#### 10.3.6.1 Interface Definitions

These interfaces basically represent the entity of the design. If the width of the interface signal is parameterized, then that information must be supplied. A sample template of the interface signals is demonstrated in table 10.3.6.1.

**Table 10.3.6.1 Sample Design Interfaces**

Port Name	Type	Size	Function
Clk	in	1	20 MHz Clock
Data	inout	Data_g downto 0 ( <i>Data_g</i> is a generic initially set to 32)	Data interface carrying memory, processor, and FPGA data.

The types of the port can include the following port characteristics:

1. <b>in</b>	Input
2. <b>out</b>	Totem Pole output
3. <b>inout</b>	Tri-State bi-directional, no pullup resistor required
4. <b>s_inout</b>	Sustained tri-state. Any agent releasing the signal to the tri-state
<b>s_out</b>	level must pre-charge the signal to a logic '1'. A pullup resistor is required to sustain the inactive state.
5. <b>o_inout</b>	Open drain. A pullup is required to sustain the inactive state.
<b>o_out</b>	

### 10.3.6.2 Block Diagrams and Registers

A block diagram of the VHDL design enhances comprehension of the architecture. This is a block diagram of the VHDL code that includes the following:

1. **Design hierarchy.** This hierarchy must include not only the hierarchical VHDL components, but also the VHDL blocks.
2. **Design registers.** All registers used in the design must be defined, using the VHDL names. Table 10.3.6.2 demonstrates a sample listing of register definitions.

**Table 10.3.6.2 Sample Register Listing**

Register	I/O	Size	Block Name	Function
<i>Instruction_r</i>	none	<i>2 ** Ins_c</i>	<i>Instruction_Blk</i>	Stores the microcode instruction
<i>DataOut_r</i>	Data	32	<i>Data_Blk</i>	Stores the <i>Data</i> output

NOTE: Sizes are defined in package XYZ\_Pkg using constants (see section 10.3)

3. **Design Functional Blocks.** Functional blocks, such as arithmetic logic units and encoders, must be defined. Figure 10.3.5-1 demonstrates an example of such block diagrams.

### 10.3.6.3 State Diagrams

If state diagrams are used in the design process, then those diagrams must be included to clarify the design.

### 10.3.6.4 Design Rationale

The rationales for many of the design decisions should be described herein. These rationales may include the design partitioning, the selection of components or primitive cores, the architectural features, the selected technology and packaging, etc.

### 10.3.6.5 Operation Modes

The modes of the design must be described (if applicable).

#### 10.3.6.6 Instruction Set Architecture (*ISA*)

If an architecture includes an *ISA*, that *ISA* must be described. For microprocessor designs, the *ISA* description can be quite extensive, and may be included in a separate document.

#### 10.3.6.7 Software Interface (Hardware/Firmware) Model

A description (or model) of all the resources and addresses accessible from software clarifies the software interface to the device. This model includes:

- Definition, function, and address of each register.
- Mode controls of the device.
- Initialization requirements to set the machine in the desired states or modes.
- Sequencing of operations to perform the desired operations.

#### 10.3.6.8 Design Restrictions and Ambiguities

All known restrictions or ambiguities must be described. These may include restrictions in addressing sequence, or interface timing ambiguities, or precharge requirements on interface signals.

#### 10.3.6.9 Model Scalability

A description of how the VHDL code can be scaled to meet changes to requirements. For each scaleable parameter, the following must be included: its location (e.g., package), its significance, and legal values. As explained in section 10.3, scalability can be achieved through the definition of generics, constants, and subtypes. Items that are typically scaled include:

1. Interface port size (or width).
2. Register size (or width).
3. Register file depth.
4. Physical addresses for each hardware resource.
5. Physical location of each of the fields in control registers (e.g., *Source* field = Bits 20 *downto* 16, *Dest* field = Bits 10 *downto* 6).

#### 10.3.6.10 Design Testability

Testability approaches used in the design must be defined. These include the following descriptions:

- Functional tests
- Acceptance tests
- In-circuit tests (e.g., Periodic tests)
- Self-tests (e.g., at power-up or upon command)
- Boundary scan (e.g., IEEE 1149.1)

### 10.3.6.11 Hardware Initialization

A description of how the design can be hard-initialized, and requirements for the initialization signals must be defined. For example, identify the number of clock cycles the *RESET* signal must be asserted to the active state. The state of the machine upon hardware initialization must also be clearly identified (e.g., all registers reset to *ZERO*, at the benign state).

### 10.3.6.12 Electrical Specification

The electrical characteristics of the device and the interfaces must be described. These parameters include power supply current, input/output current, input/output voltage specifications. A model for these specifications can be extracted from commercial specifications of components.

### 10.3.6.13 Timing specifications

Timing specifications are often referred as *AC* or *SWITCHING* characteristics. These include clocking characteristics, propagation delays, and setup and hold times. The timing specifications should also include representative timing waveforms for each of the critical scenarios (i.e., READ, WRITE cycles). A model for these specifications and timing waveforms can be extracted from commercial specifications of components. Drawing tools (such as *Chronology's* [17] *TimingDesigner* or *QuickBench*) can facilitate the generation of waveform interface specifications.

### 10.3.6.14 Design Files and compilation order

The design files necessary to compile, synthesize, and verify the design must be defined. The files must also include the complete storage path. Table 10.3.6.14 provides a sample format.

**Table 10.3.6.14 VHDL Files**

Compilation Order	Path	FILE	FUNCTION
1	/home/account/design/vhdl	xyz_pb.vhd	<ul style="list-style-type: none"> <li>• Package <i>XYZ_Pkg</i> that defines the scalable parameters</li> </ul>
2	/home/account/design/vhdl	xyz.vhd	<ul style="list-style-type: none"> <li>• Entity <i>XYZ</i></li> <li>• Architecture <i>XYZ_a</i></li> </ul>
...	....	....	<ul style="list-style-type: none"> <li>• ....</li> </ul>

### 10.3.6.15 Application notes

Any information that relates to the application of the design must be described. This may include notes that relate to the application of the component in a subsystem and its interfaces to other components.

### 10.3.7 Synthesis

Synthesis is the process of mapping a design onto a gate-level model using a targeted technology and library (i.e., behavioral or *RTL* model to gate level with registers and gates). Synthesis is a separate process than **technology mapping and routing** that uses the outputs of the synthesis process (the gate level model) to map and route the design onto a specific hardware component. Some tools separate those two processes (synthesis and hardware mapping). For example, *Synopsys* can convert an *RTL* design into a gate-level design defined in an *Electronic Design Interchange Format (EDIF)* format. The technology mapping and routing can be performed by a routing tool that reads the *EDIF* file and maps the design into the desired hardware technology. Other tools use other intermediate (internal) formats for this conversion process.

This section must document the following: tools used in the synthesis process; the directed technology used in this process; compilation duration; synthesis status; issues; libraries; compilation scripts; and compilation order. The goal is to document all the information necessary to replicate the process should the design be reused with a different technology or be modified.

#### 10.3.7.1 Synthesis tools

This section describes the synthesis tools used in the synthesis process (e.g., *Synopsys*, *Synplify by Synplicity*).

#### 10.3.7.2 Synthesis Library

This section describes the synthesis library (e.g., *Altera 10K series*).

#### 10.3.7.3 Design Constraints

This section describes the design constraints imposed on the design, such as timing constraints, area constraints, and design optimizations. If the constraints are stored in files, then those files (and their electronic path names) must be provided.

#### 10.3.7.4 Compilation Scripts

Compilation scripts or procedures (and their electronic path names) must be defined to insure repeatability in the design process.

#### 10.3.7.5 Performance

This section describes the results of compilation. In addition, statistics on compilation time must be documented. This is useful information for estimating redesign costs, and for providing some level of confidence about the expected compilation time.

### 10.3.8 Technology Mapping and Routing

The following should be documented:

1. Tools used in the process (e.g., *Altera* software).
2. Mapped device or component (e.g., EPF10K100GC503-3).
3. Packaging data (e.g., package type, packaging diagram, number of pins, etc.).
4. Number of pins used.

5. Compilation scripts or instructions (e.g. routing instructions and priorities, delay requirements).
6. Performance, such statistics on the design (e.g. number of cells, percent gate utilization, speed). In addition post-layout timing performance must be described.
7. Port to pin mapping, including the pin assignments.
8. Mapping and routing compilation times.

### 10.3.9 Verification

The design verification methods and approaches must be described (see chapter 8). These methods may include functional simulations with the use of testbenches, verification at the behavioral, *RTL*, and gate level. The following subsections must be documented.

#### 10.3.9.1 Validation plan [9].

The validation can be a referenced document, and must include:

- Tests to verify the design requirements.
- Methods used for the tests.
- Validation approaches.
- Auto-regression test approaches.

The first three bulleted items can be described using a table format as shown in figure 10.3.9.1.

**Table 10.3.9.1 Validation Plan**

REQ #	REQUIREMENT DESCRIPTION	TEST METHOD	VALIDATION METHOD
X.a	Interface	Processor BFM to Read and Write to UUT internal registers, and to program UUT into its various modes.	Verifier verifies contents of UUT internal registers through the UUT interface (i.e., Read of what was written, or exhibit of expected modes of operations)
...	...	...	...
Y.Z	AC Timing	Analysis with a static timing analyzer under best and worst case conditions.	Comparison of calculated results against required performance.

The auto-regression test approaches define the methods used to verify that subsequent forms of the same design (e.g., gate level) meet the original design definition (e.g., *RTL* model). See section 8.3 for more detailed information on auto-regression.

### 10.3.9.2 Testbench and Verifier.

This section must define the approaches used for the verification process (e.g., formal verification, simulation with a testbench, automatic verification with verifier models). The testbench model (and all its related design information) must be supplied (see section 10.3.5). Table 10.3.9.2-2 provides an example of the components that makeup the testbench. In addition, the verifier must specify the errors that it expects to uncover (as shown in table 10.3.9.2-1), and the error injection methods used to produce those errors.

**Table 10.3.9.2-1 Summary of Expected Error Detection, and Error Injection Mechanisms**

#	ERROR DETECTED	Req #	ERROR INJECTION METHOD	ERROR DETECTION METHOD
1	"UUT failed to detect a framing error"	X--	Serial data to UUT is improperly framed.	Verifier monitors injected waveforms against the UUT's responses including the status register (read through I/O).
2	"UUT latches up in XYZ mode"	X--	Pseudo-random transactions and modes covering ....	Verifier monitors expected activities on UUT's interfaces, including ....

**Table 10.3.9.2-2 Testbench Elements**

Testbench Element	Function	Status
UUT	Component instantiation	Implemented. Only one instance of the UUT was used.
---	---	---
Server	Accepts high level instructions from a client, and translates these instructions into low level interface signals to the Unit Under Test (UUT)	Implemented. All interface protocols were modeled.
Client	Provides high level Instructions to the server	Implemented 35 sets of tests ...
Verifier	Provides automatic verification of design	Implemented, but ...
MISR Block	Provides signature of UUT interfaces, and periodically stores this information onto a file. if the gate level model signatures match the RTL level model then both design are equivalent	Implemented, Signatures are periodically stored in file "mistr.txt" every 1000 clock cycle.

### **10.3.10 Simulation**

#### 10.3.10.1 Platforms

This section must define the simulation platforms and the compiler and simulator versions used in the design processes.

### **10.3.10.2 Design Files and compilation order**

All the design files, compilation order, and compilation scripts required for simulation must be identified. Table 10.3.4.14 provides a sample format.

### **10.3.10.3 Simulation Results**

This section must define the simulation results including the tests performed at each level of modeling (*RTL*, gate level) and a summary of the results. This information must include:

- Simulation run length in clock cycles, simulated time, actual wall clock time.
- Simulation coverage (or amount of UUT covered by the tests).
- A log of the detected design errors. This log must include the date of the error discoveries, the types of error, the causes of the errors, the methods of discovery, and the design resolutions.

### **10.3.11 Backup Media**

Magnetic, compact disks (CD), or other backup media of all the design files must be made with a clear identity as to where those storage media backups can be accessed.

### **10.3.12 Recommendations**

Recommendations must be provided with regard to further tests or design changes as a result of the verification analyses.

### **10.3.13 Index**

The index provides a quick reference as to where information can be found. This index is similar to the index found in a textbook.

## Appendix A : VHDL'93 AND VHDL'87 SYNTAX SUMMARY

---

---

abstract\_literal ::= decimal\_literal | based\_literal  
access\_type\_definition ::= access subtype\_indication  
actual\_designator ::=  
    expression  
    | signal\_name  
    | variable\_name  
    | file\_name  
    | open  
actual\_parameter\_part ::= parameter\_association\_list  
actual\_part ::=  
    actual\_designator  
    | function\_name ( actual\_designator )  
    | type\_mark ( actual\_designator )  
adding\_operator ::= + | - | &  
aggregate ::= ( element\_association { , element\_association } )  
alias\_declaration ::= alias alias\_designator [ : subtype\_indication ] is  
    name [ signature ] ;  
alias\_designator ::= identifier | character\_literal |  
    operator\_symbol  
allocator ::= new subtype\_indication  
    | new qualified\_expression  
architecture\_body ::= architecture identifier of entity\_name is  
    architecture\_declarative\_part  
    begin  
        architecture\_statement\_part  
    end [ architecture ] [ architecture\_simple\_name ] ;  
architecture\_declarative\_part ::= { block\_declarative\_item }  
architecture\_statement\_part ::= { concurrent\_statement }  
array\_type\_definition ::= unconstrained\_array\_definition |  
    constrained\_array\_definition  
assertion ::= assert condition  
    [ report expression ]  
    [ severity expression ]  
assertion\_statement ::= [ label : ] assertion ;  
association\_element ::= [ formal\_part => ] actual\_part  
association\_list ::= association\_element { , association\_element }  
attribute\_declaration ::= attribute identifier : type\_mark ;

```

attribute_designator ::= attribute_simple_name
attribute_name ::= 
  prefix [ signature ] ' attribute_designator
  [ (expression) ]
attribute_specification ::= 
  attribute attribute_designator of
  entity_specification is expression ;
base ::= integer
baseSpecifier ::= B | O | X
base_unit_declaration ::= identifier ;
based_integer ::= 
  extended_digit { [ underline ] extended_digit }
based_literal ::= 
  base # based_integer [ . based_integer ] #
  [ exponent ]
basic_character ::= 
  basic_graphic_character | format_effector
basic_graphic_character ::= 
  upper_case_letter | digit | special_character|
  space_character
basic_identifier ::= 
  letter { [ underline ] letter_or_digit }
binding_indication ::= 
  [ use entity_aspect ]
  [ generic_map_aspect ]
  [ port_map_aspect ]
bit_string_literal ::=  baseSpecifier " bit_value "
bit_value ::= extended_digit { [ underline ]
  extended_digit }
block_configuration ::= 
  for block_specification
    { use_clause }
    { configuration_item }
  end for ;
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
block_declarative_part ::= 
  { block_declarative_item }
block_header ::= 
  [ generic_clause
  [ generic_map_aspect ; ] ]
  [ port_clause
  [ port_map_aspect ; ] ]
block_specification ::= 
  architecture_name
  | block_statement_label
  | generate_statement_label
  [ ( index_specification ) ]
block_statement ::= 
  block_label :
    block [ ( guard_expression ) ] [ is ]
      block_header
      block_declarative_part
      begin
        block_statement_part
      end block [ block_label ];
  block_statement_part ::= 
    { concurrent_statement }
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
character_literal ::= ' graphic_character '

```

```

choice ::= simple_expression
         | discrete_range
         | element_simple_name
         | others

choices ::= choice { | choice }

component_configuration ::= -- VHDL'87
    for component_specification
        [ use binding_indication ; ]
        [ block_configuration ]
    end for ;

component_configuration ::= -- VHDL'93
    for component_specification
        [binding_indication ; ]
        [ block_configuration ]
    end for ;

component_declaration ::= component identifier [ is ]
    [ local_generic_clause ]
    [ local_port_clause ]
end component [ component_simple_name ] ;

component_instantiation_statement ::= -- VHDL'87
    instantiation_label :
        component_name
        [ generic_map_aspect ]
        [ port_map_aspect ] ;

component_instantiation_statement ::= -- VHDL'93
    instantiation_label :
        instantiated_unit
        [ generic_map_aspect ]
        [ port_map_aspect ] ;

component_specification ::= instantiation_list : component_name

composite_type_definition ::= array_type_definition
                            | record_type_definition

concurrent_assertion_statement ::= [ label : ] [ postponed ] assertion ;

concurrent_procedure_call_statement ::= [ label : ] [ postponed ] procedure_call ;

concurrent_signal_assignment_statement ::= [ label : ] [ postponed ]
    conditional_signal_assignment
    [ [ label : ] [ postponed ]
        selected_signal_assignment
    ]

```

concurrent\_statement ::= block\_statement  
 | process\_statement  
 | concurrent\_procedure\_call\_statement  
 | concurrent\_assertion\_statement  
 | concurrent\_signal\_assignment\_statement  
 | component\_instantiation\_statement  
 | generate\_statement

condition ::= boolean\_expression

condition\_clause ::= until condition

conditional\_signal\_assignment ::= target <= options  
 conditional\_waveforms ;

conditional\_waveforms ::= { waveform when condition else }  
 waveform [ when condition ]

configuration\_declaration ::= configuration identifier of entity\_name is  
 configuration\_declarative\_part  
 block\_configuration  
 end [ configuration ]  
 [ configuration\_simple\_name ] ;

configuration\_declarative\_item ::= use\_clause  
 | attribute\_specification  
 | group\_declaration

configuration\_declarative\_part ::= { configuration\_declarative\_item }

configuration\_item ::= block\_configuration  
 | component\_configuration

configuration\_specification ::= -- VHDL'87  
 for component\_specification use  
 binding\_indication ;

configuration\_specification ::= -- VHDL'93  
 for component\_specification binding\_indication ;

constant\_declaration ::= constant identifier\_list : subtype\_indication  
 [ := expression ] ;

constrained\_array\_definition ::= array index\_constraint of  
 element\_subtype\_indication

```

constraint ::= range_constraint | index_constraint
context_clause ::= { context_item }

context_item ::= library_clause | use_clause

decimal_literal ::= integer [ . integer ] [ exponent ]

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

delay_mechanism ::= -- VHDL'93
    transport | [ reject time_expression ] inertial

design_file ::= design_unit { design_unit }

design_unit ::= context_clause library_unit

designator ::= identifier | operator_symbol

direction ::= to | downto

disconnection_specification ::= disconnect guarded_signal_specification after time_expression ;

discrete_range ::= discrete_subtype_indication | range

element_association ::= [ choices => ] expression

element_declaration ::= identifier_list : element_subtype_definition ;

element_subtype_definition ::= subtype_indication

entity_aspect ::= entity entity_name [ ( architecture_identifier ) ] | configuration configuration_name | open

entity_class ::= entity | architecture | configuration | procedure | function | package | type | subtype | constant | signal | variable | component | label | literal | units | group | file

entity_class_entry ::= entity class [ <> ]

entity_class_entry_list ::= entity class entry { , entity class entry }

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

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

entity_declarative_part ::= { entity_declarative_item }

-- VHDL'87
entity_designator ::= simple_name | operator_symbol
-- VHDL'93
entity_designator ::= entity_tag [ signature ]

```

```

entity_header ::= -- VHDL'87
  [ formal_generic_clause ]
  [ formal_port_clause ]

entity_name_list ::= -- VHDL'93
  entity_designator { , entity_designator }
  | others
  | all

entity_specification ::= -- VHDL'93
  entity_name_list : entity_class

entity_statement ::= -- VHDL'93
  concurrent_assertion_statement
  | passive_concurrent_procedure_call_statement
  | passive_process_statement

entity_statement_part ::= -- VHDL'93
  { entity_statement }

entity_tag ::= simple_name | character_literal | operator_symbol

enumeration_literal ::= identifier | character_literal

enumeration_type_definition ::= -- VHDL'93
  ( enumeration_literal { , enumeration_literal } )

exit_statement ::= -- VHDL'93
  [ label ] exit [ loop_label ] [ when condition ];

exponent ::= E [ + ] integer | E - integer

expression ::= -- VHDL'93
  relation { and relation }
  | relation { or relation }
  | relation { xor relation }
  | relation { nand relation }
  | relation { nor relation }
  | relation { xnor relation }

extended_digit ::= digit | letter

extended_identifier ::= -- VHDL'93
  \graphic_character { graphic_character } \

factor ::= -- VHDL'93
  primary [ ** primary ]
  | abs primary
  | not primary

file_declaration ::= -- VHDL'87
  file identifier : subtype_indication is [ mode ]
  file_logical_name;

file_declaration ::= -- VHDL'93
  file identifier_list : subtype_indication
  file_open_information];

file_logical_name ::= string_expression

file_open_information ::= -- VHDL'93
  [ open file_open_kind_expression ] is
  file_logical_name

file_type_definition ::= -- VHDL'93
  file of type_mark

floating_type_definition ::= range_constraint

formal_designator ::= -- VHDL'93
  generic_name
  | port_name
  | parameter_name

formal_parameter_list ::= parameter_interface_list

formal_part ::= -- VHDL'93
  formal_designator
  | function_name ( formal_designator )
  | type_mark ( formal_designator )

full_type_declaration ::= -- VHDL'93
  type identifier is type_definition;

function_call ::= -- VHDL'93
  function_name ( actual_parameter_part )

generate_statement ::= -- VHDL'87
  generate_label : generation_scheme generate
  {concurrent_statement}
  end generate [ generate_label ];

generate_statement ::= -- VHDL'93
  generate_label :
  generation_scheme generate
  [ { block_declarative_item }
  begin ]
  { concurrent_statement }
  end generate [ generate_label ];

```

generation\_scheme ::=  
   for generate\_parameter\_specification  
   | if condition

generic\_clause ::=  
   generic ( generic\_list );

generic\_list ::= generic\_interface\_list

generic\_map\_aspect ::=  
   generic map ( generic\_association\_list )

graphic\_character ::=  
   basic\_graphic\_character | lower\_case\_letter |  
   other\_special\_character

group\_constituent ::= name | character\_literal

group\_constituent\_list ::= group\_constituent  
   { , group\_constituent }

group\_declaration ::=  
   group identifier : group\_template\_name  
   ( group\_constituent\_list );

group\_template\_declaration ::=  
   group identifier is ( entity\_class\_entry\_list );

guarded\_signal\_specification ::=  
   guarded\_signal\_list : type\_mark

-- VHDL'87

identifier ::=  
   letter { [ underline ] letter\_or\_digit }

-- VHDL'93

identifier ::=  
   basic\_identifier | extended\_identifier

identifier\_list ::= identifier { , identifier }

if\_statement ::=  
   [ 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 ;

index\_constraint ::= ( discrete\_range  
   { , discrete\_range } )

index\_specification ::=  
   discrete\_range  
   | static\_expression

index\_subtype\_definition ::= type\_mark range <>

indexed\_name ::= prefix ( expression { , expression } )

-- VHDL'93

instantiated\_unit ::=  
   [ component ] component\_name  
   | entity entity\_name [ ( architecture\_identifier ) ]  
   | configuration configuration\_name

instantiation\_list ::=  
   instantiation\_label { , instantiation\_label }  
   | others  
   | all

integer ::= digit { [ underline ] digit }

integer\_type\_definition ::= range\_constraint

interface\_constant\_declaration ::=  
   [ constant ] identifier\_list : [ in ]  
   subtype\_indication [ := static\_expression ]

interface\_declaration ::=  
   interface\_constant\_declaration  
   | interface\_signal\_declaration  
   | interface\_variable\_declaration  
   | interface\_file\_declaration

interface\_element ::= interface\_declaration

-- VHDL'93

interface\_file\_declaration ::=  
   file identifier\_list : subtype\_indication

interface\_list ::=  
   interface\_element { ; interface\_element }

interface\_signal\_declaration ::=  
   [ signal ] identifier\_list : [ mode ]  
   subtype\_indication [ bus ] [ := static\_expression ]

interface\_variable\_declaration ::=  
   [ variable ] identifier\_list : [ mode ]  
   subtype\_indication [ := static\_expression ]

iteration\_scheme ::=  
   while condition  
   | for loop\_parameter\_specification

```

label ::= identifier                                | variable_declaration
                                                 | file_declaration

letter ::= upper_case_letter | lower_case_letter

letter_or_digit ::= letter | digit

library_clause ::= library logical_name_list ;

library_unit ::= primary_unit
               | secondary_unit

literal ::= numeric_literal
           | enumeration_literal
           | string_literal
           | bit_string_literal
           | null

logical_name ::= identifier

logical_name_list ::= logical_name { , logical_name }

logical_operator ::= and | or | nand | nor | xor | xnor

loop_statement ::= [ loop_label : ]
                  [ iteration_scheme ] loop
                  sequence_of_statements
                  end loop [ loop_label ];

miscellaneous_operator ::= ** | abs | not

mode ::= in | out | inout | buffer | linkage

multiplying_operator ::= * | / | mod | rem

name ::= simple_name
       | operator_symbol
       | selected_name
       | indexed_name
       | slice_name
       | attribute_name

next_statement ::= [ label : ] next [ loop_label ] [ when condition ];

null_statement ::= [ label : ] null ;

numeric_literal ::= abstract_literal
                   | physical_literal

object_declaration ::= constant_declaration
                     | signal_declaration

operator_symbol ::= string_literal

-- VHDL'87
options ::= [ guarded ] [ transport ]

-- VHDL'93
options ::= [ guarded ] [ delay_mechanism ]

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

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

package_body_declarative_part ::= { package_body_declarative_item }

package_declaration ::= package identifier is
                         package_declarative_part
                         end [ package ] [ package_simple_name ];

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

```

```

package_declarative_part ::= 
    { package_declarative_item }

parameter_specification ::= 
    identifier in discrete_range

physical_literal ::= [ abstract_literal ] unit_name

physical_type_definition ::= 
    range_constraint
    units
    base_unit_declaration
    { secondary_unit_declaration }
    end units [ physical_type_simple_name ]

port_clause ::= 
    port ( port_list );

port_list ::= port_interface_list

port_map_aspect ::= 
    port map ( port_association_list )

prefix ::= 
    name
    | function_call

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

primary_unit ::= 
    entity_declaration
    | configuration_declaration
    | package_declaration

procedure_call ::= procedure_name
    [ ( actual_parameter_part ) ]

procedure_call_statement ::= 
    [ label : ] procedure_call ;

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_template_declaration
    | group_declaration

process_declarative_part ::= 
    { process_declarative_item }

process_statement ::= 
    [ process_label : ]
    [ postponed ] process [ ( sensitivity_list ) ] [ is ]
    process_declarative_part
    begin
    process_statement_part
    end [ postponed ] process [ process_label ];

process_statement_part ::= 
    { sequential_statement }

qualified_expression ::= 
    type_mark ' ( expression )
    | type_mark ' aggregate

range ::= 
    range_attribute_name
    | simple_expression direction simple_expression

range_constraint ::= range range

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

relation ::= 
    shift_expression [ relational_operator ]
    shift_expression

relational_operator ::= 
    = | /= | < | <= | > | >=

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

report_statement ::= 
    [ label : ]
    report expression
    [ severity expression ];

```

```

scalar_type_definition ::= -- VHDL'87
  enumeration_type_definition |
  integer_type_definition |
  floating_type_definition |
  physical_type_definition

secondary_unit ::= -- VHDL'93
  architecture_body |
  package_body

secondary_unit_declaration ::= signal_assignment_statement
  identifier = physical_literal;

selected_name ::= prefix . suffix
selected_signal_assignment ::= signal_declaration
  with expression select
    target <= options selected_waveforms;

selected_waveforms ::= signal_list
  { waveform when choices , }
  waveform when choices

sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name { , signal_name }

sequence_of_statements ::= simple_expression
  { sequential_statement }

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

simple_name ::= identifier
slice_name ::= prefix ( discrete_range )

string_literal ::= " { graphic_character } "

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

subprogram_declaration ::= subprogram_specification;

-- VHDL'93
shift_expression ::= subprogram_declaration
  simple_expression
  [ shift_operator simple_expression ]

-- VHDL'93
shift_operator ::= sll | srl | sla | sra | rol | ror

sign ::= + | -

```

```

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_declarative_part ::= 
    { subprogram_declarative_item }

subprogram_kind ::= procedure | function

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

subprogram_statement_part ::= 
    { sequential_statement }

subtype_declaration ::= 
    subtype identifier is subtype_indication;

subtype_indication ::= 
    [ resolution_function_name ] type_mark
    [ constraint ]

suffix ::= 
    simple_name
    | character_literal
    | operator_symbol
    | all

target ::= 
    name
    | aggregate

term ::= 
    factor { multiplying_operator factor }

timeout_clause ::= for time_expression

type_conversion ::= type_mark ( expression )

type_declaration ::= 
    full_type_declaration
    | incomplete_type_declaration

type_definition ::= 
    scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition

type_mark ::= 
    type_name
    | subtype_name

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

use_clause ::= 
    use selected_name { , selected_name };

variable_assignment_statement ::= 
    [ label : ] target := expression;

variable_declaration ::= 
    [ shared ] variable identifier_list :
        subtype_indication [ := expression ];

wait_statement ::= 
    [ label : ] wait [ sensitivity_clause ]
        [ condition_clause ] [ timeout_clause ];

waveform ::= 
    waveform_element { , waveform_element }
    | unaffected

waveform_element ::= 
    value_expression [ after time_expression ]
    | null [ after time_expression ]

```

## **Appendix B : PACKAGE STANDARD**

```
-- This is Package STANDARD as defined in the VHDL 1992 Language Reference Manual.
-- Reprinted by permission from Model Technology Inc.
-- NOTE: VCOM and VSIM will not work properly if these declarations
--       are modified.

-- Version information: @(#)standard.vhd

package standard is
    type boolean is (false,true);
    type bit is ('0', '1');
    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,
        -- the character code for 160 is there (NBSP),
        -- but prints as no char
        ' ', ' ; ', '¢ ', '£ ', '¤ ', '¥ ', ' ', ' ¢ ',
        ' ', '® ', '™ ', '« ', '» ', '¬ ', '® ', '¬ ',
        '° ', '± ', '² ', '³ ', '° ', 'µ ', '¶ ', '° ',
        '° ', '° ', '° ', '° ', '° ', '° ', '° ', '° '
```

```
'À', 'Á', 'Â', 'Ã', 'Ä', 'È', 'É', 'Ê', 'Í', 'Ï', 'Ó', 'Ñ', 'Ô', 'Ö', '×',  
'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',  
  
'à', 'á', 'â', 'ã', 'ä', 'è', 'é', 'ê', 'í', 'ï', 'ó', 'ñ', 'ô', 'ö', '÷',  
'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ');  
  
type severity_level is (note, warning, error, failure);  
type integer is range -2147483647 to 2147483647; -- per LRM minimum range  
type real is range -1.0E308 to 1.0E308;  
type time is range -2147483647 to 2147483647 -- per LRM minimum range  
    units  
        fs;  
        ps = 1000 fs;  
        ns = 1000 ps;  
        us = 1000 ns;  
        ms = 1000 us;  
        sec = 1000 ms;  
        min = 60 sec;  
        hr = 60 min;  
    end units;  
subtype delay_length is time range 0 fs to time'high;  
impure function now return delay_length;  
subtype natural is integer range 0 to integer'high;  
subtype positive is integer range 1 to integer'high;  
type string is array (positive range <>) of character;  
type bit_vector is array (natural range <>) of bit;  
type file_open_kind is (  
    read_mode,  
    write_mode,  
    append_mode);  
type file_open_status is (  
    open_ok,  
    status_error,  
    name_error,  
    mode_error);  
    attribute foreign : string;  
end standard;
```

## **Appendix C : PACKAGE TEXTIO**

---

---

```
-----  
-- Package TEXTIO as defined in Chapter 14 of the IEEE Standard VHDL  
-- Language Reference Manual (IEEE Std. 1076-1987), as modified  
-- by the Issues Screening and Analysis Committee (ISAC), a subcommittee  
-- of the VHDL Analysis and Standardization Group (VASG) on  
-- 10 November, 1988. See "The Sense of the VASG", October, 1989.  
-- Reprinted by permission from Model Technology Inc.  
-----  
-- Version information: %W% %G%  
-----  
  
package TEXTIO is  
    type LINE is access string;  
    type TEXT is file of string;  
    type SIDE is (right, left);  
    subtype WIDTH is natural;  
  
    -- changed for vhdl92 syntax:  
    file input : TEXT open read_mode is "STD_INPUT";  
    file output : TEXT open write_mode is "STD_OUTPUT";  
  
    -- changed for vhdl92 syntax (and now a built-in):  
    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);

-- changed for vhdl92 syntax (and now a built-in):
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);

-- is implicit built-in:
-- function ENDFILE(file F : TEXT) return boolean;

-- function ENDLINE(variable L : in LINE) return BOOLEAN;
--
-- Function ENDLINE as declared cannot be legal VHDL, and
-- the entire function was deleted from the definition
-- by the Issues Screening and Analysis Committee (ISAC),
-- a subcommittee of the VHDL Analysis and Standardization
-- Group (VASG) on 10 November, 1988. See "The Sense of
-- the VASG", October, 1989, VHDL Issue Number 0032.

end;

*****  

***  

*** Copyright (c) Model Technology Incorporated 1991 **  

*** All Rights Reserved **  

***  

*****

```

## **Appendix D : PACKAGE STD\_LOGIC\_1164**

---

---

```
-- -----
-- Title      : std_logic_1164 multi-value logic system
-- Library    : This package shall be compiled into a library
--               : symbolically named IEEE.
--               :
-- Developers : IEEE model standards group (par 1164)
-- Purpose    : This packages defines a standard for designers
--               : to use in describing the interconnection data types
--               : used in vhdl modeling.
--               :
-- Limitation: The logic system defined in this package may
--               : be insufficient for modeling switched transistors,
--               : since such a requirement is out of the scope of this
--               : effort. Furthermore, mathematics, primitives,
--               : timing standards, etc. are considered orthogonal
--               : issues as it relates to this package and are therefore
--               : beyond the scope of this effort.
--               :
-- Note       : No declarations or definitions shall be included in,
--               : or excluded from this package. The "package declaration"
--               : defines the types, subtypes and declarations of
--               : std_logic_1164. The std_logic_1164 package body shall be
--               : considered the formal definition of the semantics of
--               : this package. Tool developers may choose to implement
--               : the package body in the most efficient manner available
--               : to them.
--               :
-- modification history :
-- -----
-- version | mod. date:|
-- v4.200 | 01/02/92 |
```

```
PACKAGE std_logic_1164 IS
  -- logic state system  (unresolved)
  --
  TYPE std_ulogic IS ( 'U',  -- Uninitialized
                       'X',  -- Forcing Unknown
                       '0',  -- Forcing 0
                       '1',  -- Forcing 1
                       'Z',  -- High Impedance
                       'W',  -- Weak Unknown
```

```

        'L',  -- Weak      0
        'H',  -- Weak      1
        '-'   -- Don't care
    );
-- unconstrained array of std_ulogic for use with the resolution function
-- TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

-- resolution function
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

-- *** industry standard logic type ***
SUBTYPE std_logic IS resolved std_ulogic;

-- unconstrained array of std_logic for use in declaring signal arrays
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_logic;

-- common subtypes
SUBTYPE X01      IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
SUBTYPE X01Z     IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
SUBTYPE UX01     IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')
SUBTYPE UX01Z    IS resolved std_ulogic RANGE 'U' TO 'Z';
-- ('U','X','0','1','Z')

-- overloaded logical operators
FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "or"    ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nor"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xor"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xnor"  ( l : std_ulogic; r : std_ulogic ) RETURN ux01;
FUNCTION "not"   ( l : std_ulogic           ) RETURN UX01;

-- vectorized overloaded logical operators
FUNCTION "and"  ( l, r : std_logic_vector ) RETURN std_logic_vector;
FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "nand" ( l, r : std_logic_vector ) RETURN std_logic_vector;
FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "or"   ( l, r : std_logic_vector ) RETURN std_logic_vector;
FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "nor"  ( l, r : std_logic_vector ) RETURN std_logic_vector;
FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "xor"  ( l, r : std_logic_vector ) RETURN std_logic_vector;

```

```

FUNCTION "xor" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
-- Note : The declaration and implementation of the "xnor" function is
-- specifically commented until at which time the VHDL language has been
-- officially adopted as containing such a function. At such a point,
-- the following comments may be removed along with this notice without
-- further "official" balloting of this std_logic_1164 package. It is
-- the intent of this effort to provide such a function once it becomes
-- available in the VHDL standard.
-- function "xnor" ( l, r : std_logic_vector ) return std_logic_vector;
-- function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;

FUNCTION "not" ( l : std_logic_vector ) RETURN std_logic_vector;
FUNCTION "not" ( l : std_ulogic_vector ) RETURN std_ulogic_vector;

-- conversion functions

FUNCTION To_bit      ( s : std_ulogic;           xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0')
                      RETURN BIT_VECTOR;
FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0')
                      RETURN BIT_VECTOR;

FUNCTION To_StdULogic   ( b : BIT                  ) RETURN std_ulogic;
FUNCTION To_StdLogicVector ( b : BIT_VECTOR          ) RETURN std_logic_vector;
FUNCTION To_StdLogicVector ( s : std_ulogic_vector ) RETURN std_logic_vector;
FUNCTION To_StdULogicVector ( b : BIT_VECTOR ) RETURN std_ulogic_vector;
FUNCTION To_StdULogicVector ( s : std_logic_vector) RETURN std_ulogic_vector;

-- strength strippers and type converters

FUNCTION To_X01   ( s : std_logic_vector ) RETURN std_logic_vector;
FUNCTION To_X01   ( s : std_ulogic_vector ) RETURN std_ulogic_vector;
FUNCTION To_X01   ( s : std_ulogic        ) RETURN X01;
FUNCTION To_X01   ( b : BIT_VECTOR         ) RETURN std_logic_vector;
FUNCTION To_X01   ( b : BIT_VECTOR         ) RETURN std_ulogic_vector;
FUNCTION To_X01   ( b : BIT              ) RETURN X01;

FUNCTION To_X01Z  ( s : std_logic_vector ) RETURN std_logic_vector;
FUNCTION To_X01Z  ( s : std_ulogic_vector ) RETURN std_ulogic_vector;
FUNCTION To_X01Z  ( s : std_ulogic        ) RETURN X01Z;
FUNCTION To_X01Z  ( b : BIT_VECTOR         ) RETURN std_logic_vector;
FUNCTION To_X01Z  ( b : BIT_VECTOR         ) RETURN std_ulogic_vector;
FUNCTION To_X01Z  ( b : BIT              ) RETURN X01Z;

FUNCTION To_UX01  ( s : std_logic_vector ) RETURN std_logic_vector;
FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN std_ulogic_vector;
FUNCTION To_UX01  ( s : std_ulogic        ) RETURN UX01;
FUNCTION To_UX01  ( b : BIT_VECTOR         ) RETURN std_logic_vector;
FUNCTION To_UX01  ( b : BIT_VECTOR         ) RETURN std_ulogic_vector;
FUNCTION To_UX01  ( b : BIT              ) RETURN UX01;

-- edge detection

FUNCTION rising_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;

-- object contains an unknown

FUNCTION Is_X ( s : std_ulogic_vector ) RETURN BOOLEAN;
FUNCTION Is_X ( s : std_logic_vector ) RETURN BOOLEAN;
FUNCTION Is_X ( s : std_ulogic        ) RETURN BOOLEAN;

END std_logic_1164;

```

## **Appendix E : PACKAGE STD\_LOGIC\_ARITH**

```
-- Copyright (c) 1990,1991,1992 by Synopsys, Inc. All rights reserved. --
-- This source file may be used and distributed without restriction --
-- provided that this copyright statement is not removed from the file --
-- and that any derivative work contains this copyright notice. --
-- Package name: STD_LOGIC_ARITH --
-- Purpose: --
-- A set of arithmetic, conversion, and comparison functions --
-- for SIGNED, UNSIGNED, SMALL_INT, INTEGER, --
-- STD_ULOGIC, STD_LOGIC, and STD_LOGIC_VECTOR. --
--
```

```
library IEEE;
use IEEE.std_logic_1164.all;

package std_logic_arith is
    type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
    type SIGNED is array (NATURAL range <>) of STD_LOGIC;
    subtype SMALL_INT is INTEGER range 0 to 1;

    function "+"(L: UNSIGNED; R: UNSIGNED)      return UNSIGNED;
    function "+"(L: SIGNED; R: SIGNED)           return SIGNED;
    function "+"(L: UNSIGNED; R: SIGNED)          return SIGNED;
    function "+"(L: SIGNED; R: UNSIGNED)          return SIGNED;
    function "+"(L: UNSIGNED; R: INTEGER)         return UNSIGNED;
    function "+"(L: INTEGER; R: UNSIGNED)          return UNSIGNED;
    function "+"(L: SIGNED; R: INTEGER)            return SIGNED;
    function "+"(L: INTEGER; R: SIGNED)             return SIGNED;
```

<b>function "+"(L: UNSIGNED; R: STD_ULOGIC)</b>	<b>return UNSIGNED;</b>
<b>function "+"(L: STD_ULOGIC; R: UNSIGNED)</b>	<b>return UNSIGNED;</b>
<b>function "+"(L: SIGNED; R: STD_ULOGIC)</b>	<b>return SIGNED;</b>
<b>function "+"(L: STD_ULOGIC; R: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "+"(L: UNSIGNED; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: SIGNED; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: UNSIGNED; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: SIGNED; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: UNSIGNED; R: INTEGER)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: INTEGER; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: SIGNED; R: INTEGER)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: INTEGER; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: UNSIGNED; R: STD_ULOGIC)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: STD_ULOGIC; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: SIGNED; R: STD_ULOGIC)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: STD_ULOGIC; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: UNSIGNED; R: UNSIGNED)</b>	<b>return UNSIGNED;</b>
<b>function "-"(L: SIGNED; R: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "-"(L: UNSIGNED; R: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "-"(L: SIGNED; R: UNSIGNED)</b>	<b>return SIGNED;</b>
<b>function "-"(L: UNSIGNED; R: INTEGER)</b>	<b>return UNSIGNED;</b>
<b>function "-"(L: INTEGER; R: UNSIGNED)</b>	<b>return UNSIGNED;</b>
<b>function "-"(L: SIGNED; R: INTEGER)</b>	<b>return SIGNED;</b>
<b>function "-"(L: INTEGER; R: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "-"(L: UNSIGNED; R: STD_ULOGIC)</b>	<b>return UNSIGNED;</b>
<b>function "-"(L: STD_ULOGIC; R: UNSIGNED)</b>	<b>return UNSIGNED;</b>
<b>function "-"(L: SIGNED; R: STD_ULOGIC)</b>	<b>return SIGNED;</b>
<b>function "-"(L: STD_ULOGIC; R: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "-"(L: UNSIGNED; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: SIGNED; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: UNSIGNED; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: SIGNED; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: UNSIGNED; R: INTEGER)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: INTEGER; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: SIGNED; R: INTEGER)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: INTEGER; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: UNSIGNED; R: STD_ULOGIC)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: STD_ULOGIC; R: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: SIGNED; R: STD_ULOGIC)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "-"(L: STD_ULOGIC; R: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: UNSIGNED)</b>	<b>return UNSIGNED;</b>
<b>function "+"(L: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "-"(L: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "ABST"(L: SIGNED)</b>	<b>return SIGNED;</b>
<b>function "+"(L: UNSIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>
<b>function "+"(L: SIGNED)</b>	<b>return STD_LOGIC_VECTOR;</b>

<b>function</b> "-"(L: SIGNED)	return STD_LOGIC_VECTOR;
<b>function</b> "ABS"(L: SIGNED)	return STD_LOGIC_VECTOR;
<b>function</b> ""(L: UNSIGNED; R: UNSIGNED)	return UNSIGNED;
<b>function</b> ""(L: SIGNED; R: SIGNED)	return SIGNED;
<b>function</b> ""(L: SIGNED; R: UNSIGNED)	return SIGNED;
<b>function</b> ""(L: UNSIGNED; R: SIGNED)	return SIGNED;
<b>function</b> ""(L: UNSIGNED; R: UNSIGNED)	return STD_LOGIC_VECTOR;
<b>function</b> ""(L: SIGNED; R: SIGNED)	return STD_LOGIC_VECTOR;
<b>function</b> ""(L: SIGNED; R: UNSIGNED)	return STD_LOGIC_VECTOR;
<b>function</b> ""(L: UNSIGNED; R: SIGNED)	return STD_LOGIC_VECTOR;
<b>function</b> "<"(L: UNSIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> "<"(L: SIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> "<"(L: UNSIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> "<"(L: SIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> "<"(L: UNSIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> "<"(L: INTEGER; R: UNSIGNED)	return BOOLEAN;
<b>function</b> "<"(L: SIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> "<"(L: INTEGER; R: SIGNED)	return BOOLEAN;
<b>function</b> "<="(L: UNSIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> "<="(L: SIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> "<="(L: UNSIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> "<="(L: SIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> "<="(L: UNSIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> "<="(L: INTEGER; R: UNSIGNED)	return BOOLEAN;
<b>function</b> "<="(L: SIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> "<="(L: INTEGER; R: SIGNED)	return BOOLEAN;
<b>function</b> ">"(L: UNSIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> ">"(L: SIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> ">"(L: UNSIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> ">"(L: SIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> ">"(L: UNSIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> ">"(L: INTEGER; R: UNSIGNED)	return BOOLEAN;
<b>function</b> ">"(L: SIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> ">"(L: INTEGER; R: SIGNED)	return BOOLEAN;
<b>function</b> ">="(L: UNSIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> ">="(L: SIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> ">="(L: UNSIGNED; R: SIGNED)	return BOOLEAN;
<b>function</b> ">="(L: SIGNED; R: UNSIGNED)	return BOOLEAN;
<b>function</b> ">="(L: UNSIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> ">="(L: INTEGER; R: UNSIGNED)	return BOOLEAN;
<b>function</b> ">="(L: SIGNED; R: INTEGER)	return BOOLEAN;
<b>function</b> ">="(L: INTEGER; R: SIGNED)	return BOOLEAN;
<b>function</b> "="(L: UNSIGNED; R: UNSIGNED)	return BOOLEAN;

```

function "="(L: SIGNED; R: SIGNED)      return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED)     return BOOLEAN;
function "="(L: SIGNED; R: UNSIGNED)     return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER)    return BOOLEAN;
function "="(L: INTEGER; R: UNSIGNED)    return BOOLEAN;
function "="(L: SIGNED; R: INTEGER)       return BOOLEAN;
function "="(L: INTEGER; R: SIGNED)       return BOOLEAN;

function "/="(L: UNSIGNED; R: UNSIGNED)   return BOOLEAN;
function "/="(L: SIGNED; R: SIGNED)         return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED)       return BOOLEAN;
function "/="(L: SIGNED; R: UNSIGNED)       return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER)      return BOOLEAN;
function "/="(L: INTEGER; R: UNSIGNED)      return BOOLEAN;
function "/="(L: SIGNED; R: INTEGER)        return BOOLEAN;
function "/="(L: INTEGER; R: SIGNED)        return BOOLEAN;

function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
function SHL(ARG: SIGNED; COUNT: UNSIGNED)   return SIGNED;
function SHR(ARG: UNSIGNED; COUNT: UNSIGNED)   return UNSIGNED;
function SHR(ARG: SIGNED; COUNT: UNSIGNED)   return SIGNED;

function CONV_INTEGER(ARG: INTEGER)           return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED)          return INTEGER;
function CONV_INTEGER(ARG: SIGNED)            return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC)        return SMALL_INT;

function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return UNSIGNED;

function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return SIGNED;

function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
                    return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED; SIZE: INTEGER)
                    return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE: INTEGER)
                    return STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE: INTEGER)
                    return STD_LOGIC_VECTOR;

```

-- zero extend STD\_LOGIC\_VECTOR (ARG) to SIZE,

```
-- SIZE < 0 is same as SIZE = 0
-- returns STD_LOGIC_VECTOR(SIZE-1 downto 0)
function EXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;

-- sign extend STD_LOGIC_VECTOR (ARG) to SIZE,
-- SIZE < 0 is same as SIZE = 0
-- return STD_LOGIC_VECTOR(SIZE-1 downto 0)
function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;

end Std_logic_arith;
```

## Appendix F :VHDL PREDEFINED ATTRIBUTES

---



---

### VHDL Attributes

<b>Attribute</b>	<b>Prefix</b>	<b>Comments</b>
T'base	Type	Base type of T. Must be prefix to another attribute
T'left	scalar	The left bound of T, result of type T
T'right	type/ST	The right bound of T, result of type T
T'high	scalar	The upper bound of T, result of type T
T'low	type/ST	The lower bound of T, result of type T
T'Ascending VHDL'93	scalar type/ST	TRUE if type T is ascending
T'image(X) VHDL'93	scalar type/ST	Function which converts scalar object X of type T into string
T'value(X) VHDL'93	scalar type/ST	Function which converts object X of type string into scalar of type T
T'pos(X)	discrete /PT/ST	Function which returns a universal integer representing the position number of parameter X of type T. First position = 0.
T'val(X)	discrete /PT/ST	Function which returns of base type T the value whose position is the universal integer value corresponding to X.
T'succ(X)	discrete /PT/ST	Function returning a value of type T whose value is the position number one greater than the one of the parameter. It is an error if X = T'high or if does not belong to the range T'low to T'high
T'pred(X)	discrete /PT/ST	Function returning a value of type T whose value is the position number one less than the one of the parameter. It is an error if X = T'low or if does not belong to the range T'low to T'high
T'leftof(X)	discrete /PT/ST	Function which returns the value that is to the left of parameter X of type T. Result type is of type T. Error if X = T'left

T'rightof(X)	discrete /PT/ST	Function which returns the value that is to the right of parameter X of type T. Result type is of type T. Error is X = T'right
A'left(N)	Array*	Function which returns the left bound of the Nth index range of A. X is of type universal integer. Result type is of type of the left bound of the left index range of A. N = 1 if omitted.
A'right(A)	Array*	Same as A'left(N), except right bound is returned
A'high(N)	Array*	Function which returns the upper bound of the range of A. Result type is the type of the Nth index range of A. N = 1 if omitted.
A'low(N)	Array*	Same as A'high(N), e lower bound is returned.
A'range(N)	Array*	The range of A'left(N) to A'right(N)
A'reverse range(N)		The range of A'right(N) to A'left(N)
A'length	Array*	returns 0 if array is null. Else, returns T'pos(A'high(N)) - T'pos(A'low(N)) where T is the subtype of the Nth index of A.
A'Ascending	Array*	True if Nth index range of A is defined in an ascending range, else returns false.

PT = physical type

ST = Subtype

Array\* = Any prefix that is appropriate for an array object, (e.g. type, variable, signal) or alias thereof, or that denotes a constrained array subtype

### Summary of the VHDL Signal Attributes

S'event	Function returning a Boolean which identifies if signal S has a new value assigned onto this signal (i.e. value is different than last value).  <b>if</b> Clk'event <b>then</b> -- if Clk just changed in value then ... .... <b>wait until</b> Clk'event <b>and</b> Clk = '1'; -- rising edge of clock
S'active	Function returning a Boolean which identifies if signal S had a new assignment made onto it (whether the value of the assignment is the SAME or DIFFERENT).  <b>if</b> Data'active <b>then</b> -- New assignment of Data
S'transaction	<b>Implicit signal</b> of type bit which is created for signal S when it S'transaction is used in the code. This implicit signal is NOT declared since it is implicitly defined. This signal toggles in value (between '0' and '1') when signal S had a new assignment made onto it (whether the value of the assignment is the SAME or DIFFERENT). The user should NOT rely on its VALUE.  <b>wait on</b> ReceivedData'transaction; -- process is -- sensitive to ReceiveData rewriting any value

S'delayed(T)	<p><u>Implicit signal</u> of the same base type as S. It represents the value of signal S delayed by a time Tn. Thus, the value of S'delayed(T) at time Tn is always equal to the value of S at time Tn -t. For example, the value of S'delayed(5 ns) at time 1000 ns is the value of S at time 995 ns. Note if time is omitted, it defaults to 0 ns.</p> <pre> <b>wait on</b> Data'transaction; <b>case</b> BV2'(Data'Delayed &amp; Data) <b>is</b> -- Data @ last delta time   <b>when</b> "X0" =&gt; ... -- from X to 0 transition   <b>when</b> "10" =&gt; ... -- from 1 to 0 transition   <b>when others</b> =&gt; ... <b>end case;</b></pre>
S'stable(T)	<p><u>Implicit signal</u> of Boolean type. This implicit signal is true when an event (change in value) has NOT occurred on signal S for T time units, and the value FALSE otherwise. If time is omitted, it defaults to 0 ns.</p> <pre> <b>if</b> Data'stable(40 ns) <b>then</b> -- met set up time</pre>
S'quiet(T)	<p><u>Implicit signal</u> of Boolean type. This implicit signal is true when the signal has been quiet (i.e. no activity or signal assignment) for T time units, and the value FALSE otherwise. If time is omitted, it defaults to 0 ns.</p> <pre> <b>if</b> Data'quiet(40 ns) <b>then</b> -- Really quiet, not even an assignment of   -- the same value during the last T time units (40 ns in this example)</pre>
S'last_event	<p>The amount of time that has elapsed since the last event (change in value) occurred on signal S. If there was no previous event, it returns Time'high.</p> <pre> <b>variable</b> : TsinceLastEvent : time;   ... <b>TsinceLastEvent := Data'last_event;</b></pre>
S'last_active	<p>The amount of time that has elapsed since the last activity (assignment) occurred on signal S. If there was no previous activity, it returns Time'high.</p> <pre> <b>variable</b> : TsinceLastActivity : time;   ... <b>TsinceLastActivity := Data'last_active;</b></pre>
S'last_value	<p>Function of the base type of S returning the previous value of S, immediately before the last change of S.</p> <pre> <b>wait on</b> Data'transaction; <b>case</b> BV2'(Data'last_value &amp; Data) <b>is</b>   <b>when</b> "X0" =&gt; ... -- from X to 0 transition   <b>when</b> "10" =&gt; ... -- from 1 to 0 transition   <b>when</b> "00" =&gt; ... -- Activity, but no Data change   <b>when others</b> =&gt; ... <b>end case;</b></pre>

# BIBLIOGRAPHY

---

---

- [1] **IEEE**, *IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual*, Copyright © 1994, Institute of Electrical and Electronics Engineers, Inc. Web site: <http://stdsbsbbs.ieee.org/>
- [2] **Sandi Habinc**, *VHDL Models for Board-Level Simulation*, European Space Agency, 1996, available via *ftp* from *ftp.estec.esa.nl/pub/vhdl/doc/BoardLevel.ps*
- [3] **Cadence Design Systems, Inc.**, HDL Design Group, 270 Billerica Road, Chelmsford MA 01824, Web site: <http://www.cadence.com/>
- [4] **Alternative System Concepts, Inc.**, PO Box 128 Windham NH 03087, email: [info@ascinc.com](mailto:info@ascinc.com)
- [5] **Scientific and Engineering Software, Inc.**, SES Workbench system simulator, 4301 Westbank Drive, Blg. A, Austin, TX 78746, Web site: <http://www.ses.com>
- [6] **Model Technology Incorporated**, Vendor of *Vsystem* VHDL simulator for PC and workstations, 8905 SW Nimbus Av., Suite 150. Beaverton OR 97008-7100, Web site: <http://www.model.com>
- [7] **David Naiditch**, *RendezVous with Ada 95*, John Wiley and Sons Inc. Copyright © 1995 by David J. Naiditch, Reprinted by permission of John Wiley & Sons, Inc.
- [8] **Daniel S. Barclay**, Compass Design Automation, Inc.
- [9] **Ben Cohen**, *VHDL Coding Styles and Methodologies*, ISBN 0-7923-9598-0 Kluwer Academic Publishers, 1995, Web site: [members.aol.com/vhdlcohen](http://members.aol.com/vhdlcohen)
- [10] **Paul H. Bardell, William H. McAnney, Jacob Savir**, *Built-in Test for VLSI: Pseudorandom Techniques*, John Wiley and Sons, 1987.
- [11] **Joseph Pick of Synopsys, Inc.**, *VHDL Synthesis Techniques and Recommendations*, presented at the 1996 *Synopsys Users Group Conference*
- [12] **QuickLogic Corporation**, 2933 Bunker Hill Lane, Santa Clara, CA 95054, email: [info@qlogic.com](mailto:info@qlogic.com)
- [13] **Synplicity, Inc.**, 465 Fairchild Drive, Suite 115, Mountain View, CA 94043
- [14] **Synopsys**, *VHDL Compiler Reference Manual*, Document Order Number: 1US01-10430, Web site: <http://www.synopsys.com>
- [15] **COMPASS Design Automation**, 5457 Twin Knolls Road, Suite 100, Columbia, MD 21045, Web site: <http://www.compass-da.com>
- [16] **S. Habinc and P. Sinander**, *Using VHDL for Board Level Simulation*, IEEE Design & Test of Computers, Vol. 13, No. 3, Fall 1996, pp. 66-78;  
Web site: <http://www.computer.org/pubs/d&t/d&t.htm>
- [17] **Chronology, Timing Designer and QuickBench**, [sales@chronology.com](mailto:sales@chronology.com), URL: [\(206\)869-4227](http://www.chronology.com).

# INDEX

---

---

## A

### Access Type

- Application example, 154
- Example, 107
- Garbage collection, 106
- Linked list
  - Example, 154

### Activity

- Resolved signal, 85

### Aggregate

- Synthesis, 185
- Choices, 303

### Alias

- disk\element\cntlr\_e.vhd, 45
- Fields
  - Std\_Lo<sub>gic</sub>\_Vector, 44
- Synthesis, 184

### Array

- Aggregate, 57, 75
- Aggregate, 74, 76
- Allowed objects, 73
- Anonymous, 78
- Assignment, 58
- Concatenation, 58
- Constrained, 71
- Constraining methods, 72
- Constraining methods, 79
- Definition, 71
- Equality, 58
- File
  - Emulation, 114
- Illegal declarations, 78
- Inequality, 58
- Initialization
  - Error, 67
- Logical operators, 59
- Mapping, 74
  - disk\element\multidm.vhd, 75
  - disk\element\multidm.vhd, 74
- Multidimensional
  - Mapping, 74
- Normalization, 118
- Null, 65
  - Synthesis, 206
- One-dimentional, 56
- Operations
  - Overloaded, 60
- Operations
  - disk\element\array1.vhd, 59

### Operators

- Implicit, 58
- disk\element\signed.vhd, 61
- Others, 76
- Range definition, 303
- Relational, 58
- Resolution function
  - disk\drivers\atomic\_pb.vhd, 86
- Shift operators, 59
- Slice, 65
- Slicing, 58
- Structure representations, 56, 186
- Synthesis, 73
- Table lookup, 56
- Two-dimensional, 56
  - disk\element\tmspec.vhd, 57
- type, 71
- Type conversion, 59
- Unconstrained, 71
  - Illegal, 78
  - Others
    - Disk\element\unconsok.vhd, 76

### Asynchronous reset

- Synthesis:, 191

### Atomicity

- Drivers, 84

### Attribute

- 'Val, 47
- 'Image, 126
- 'Quiet
  - Application, 161
- 'transaction
  - Application, 159
- 'Val
  - disk\element\attribute.vhd, 48
- Enum\_Encoding, 47
- Exam, 302, 304, 328
- Image, 122, 126
- predefined (LRM 14.1), 302
- Reuse, 302, 328
- Synthesis
  - Supported/Unsupported, 185

### Auto-Regression

- See Regression/Verification, 246

## B

### Barrel shifter

- Synthesis/Behavioral, 205

### Behavioral Synthesis

- , 243

- Behavioral/RTL Modeling**, 296
- BFM (Bus Functional Model)**
- Architectural command, 255
  - Definition, 254
  - Instruction file command, 255
  - Memory, 146
  - Verification
  - Application, 254
- Binding**, 24
- Default, 25
- Bit string literal**, 44
- Block**
- Example, 11
  - Reuse, 323
  - Synthesis, 323
    - Advantages, 10
    - Restrictions, 185
  - Statement
    - Guarded in Synthesis, 17
    - Salient points, 10
    - Synthesis, 10
    - Reusability, 323
- Buffer Port**
- disk\element\buffer2.vhd, 18
  - disk\element\buffstop2.vhd, 19
  - disk\element\buffstop3.vhd, 19
  - InOut Port, 17
- C**
- Case**, 209
- & operator, 42
  - Choice, 70
  - Globally static
    - restrictions, 319
  - Priority, 209
  - Don't care, 207
- Casting**, 39, 44
- Client/server**
- disk\subprgm\drvproc.vhd, 105
  - Side Effect
    - Avoidance method, 103
- Code**
- Non-portable, 92
- Coding style**
- for design reuse, 316
- Compilation order**, 298
- Component**
- Removal, 31
  - Salient points, 8
- Concatenation**, 43
- Procedure call, 230
- Concurrent Assertion**
- Salient Points, 9
- Concurrent procedure**
- Salient points, 8
  - Side effect, 8
- Concurrent Signal assignment**
- for Waveforms, 8
  - Salient points, 7
- Concurrent statement**
- Simulation speed, 273
  - Synthesis, 185
  - Definitions, 6
  - Exam, 309
- Conditional Signal Assignment**
- Example, 7
- Configuration**
- Binding with configured components, 31
  - Configuration Declaration, 28
  - declaration, 48
    - Configured Components
      - disk\element\struct3.vhd, 31
    - Deferred Binding
      - disk\element\deferd\_c.vhd, 32
    - disk\element\cfgdecl.vhd, 29
    - disk\element\cntlr\_c.vhd, 49
    - Generate statement
      - disk\element\generat\_c.vhd, 35
    - Generate statements, 33
    - Generics, 32
      - disk\element\filter2.vhd, 33
    - Hierarchical
      - disk\element\hierarch.vhd, 30
    - Declaration/Specification, 25
    - Specification
      - disk\element\cfgspec.vhd, 27
    - Requirements, 24
    - Synthesis, 185

- Constant**  
disk\element\constant.vhd, 45  
**Fields**  
Std\_Logic\_Vector, 44  
Simulation speed, 270
- Constructs**  
Synthesis  
Supported/Unsupported, 184
- Control**  
Parameterization, 319
- Conversion**  
Hexadecimal, 122  
Integer to time, 116  
Type, 39  
Typed objects to string, 122
- Counter**  
Design  
Exam, 306  
disk  
element\count\_ea.vhd, 41  
Model, 287
- D**
- Data reduction**  
MISR/Verifier, 130
- Deallocate**, 106  
disk\subprgm\accessok.vhd, 109  
disk\subprgm\image.vhd, 107
- Declaration**  
Illegal reference, 67
- Delta time**  
Restrictions on model, 162
- Demultiplexer**  
Synthesis, 203
- Design**  
Processes, 314  
Units  
Exam, 297  
Verification, 246, 248, 251
- DesignWare**  
Reuse, 329  
Synthesis, 234
- Disconnect**, 17
- Documentation**, 337  
Applicable documents, 331  
Application notes, 335  
Definitions, 331  
Design description, 332  
Design files, 335  
Design team, 331  
Designs, 331  
Diagram, 333  
Electrical specification, 335  
Initialization, 335  
Interface, 332  
ISA, 334  
Mapping and routing, 336  
Modes, 333  
Rationale, 333  
Requirement/design matrix, 332  
Resuability, 331  
Restrictions, 334  
Scaleability, 334  
Simulation, 338  
Software model, 334  
Summary, 331  
Synthesis, 336  
Testability, 334  
Testbench, 338  
Timing specification, 335  
Verification, 337
- Don't care**  
Numeric\_Std  
Std\_Match, 211  
Synthesis  
Guideline, 207, 212
- Drivers**  
Atomicity, 84  
Creation, 82  
Exam, 309  
Initial value, 83  
Initialization, 83  
LRM 12.6.1, 83  
LRM 4.3.1.2, 84  
Multiple, 82  
    Composite, 91  
    Source, 93  
    Static, 92  
Procedure, 100  
Resolved signal, 85  
Static, 82, 83  
    Static expression, 92  
Subelement, 82

**E****EDIF**

Synthesis, 336

**Editor**

VHDL, 294

**Elements**

Inter-relationships, 2, 3

**Encoding**

One-hot, 43

**Entity**

Synthesis, 184

**Enumeration**

Conversion

disk\element\convsynt.vhd, 47

Encoding, 43

Fields, 46

Synthesis, 47

**Error**

'Image, 129

Alias

Synthesis, 119

Array

Anonymous, 78

Array

Initialization, 67

Null slice, 66

Unconstrained, 78

Association

Ground/VCC, 239

Buffer port, 19

Case

Choice, 70

Expression, 42

Concatenation operator, 42

Deallocation, 106, 107

Declaration

Identifier, 67

Drivers

Multiple, 82, 91, 92, 93

Function parameters, 110

Incompatible array types, 72

Inertial signal assignment, 227

Integer

Range, 38

Others

Unconstrained array, 75

**Slice**

Null, 66

Std\_Logic\_1164 package '87/'93, 43

Subprogram

Normalization, 118

Signal attributes, 110

Subprogram overloading, 64

Synthesis

Dynamic value, 206

Initialization, 194, 195

Null slice, 205

Type declaration

Multiple declarations, 293

Types, 39

Unconstrained Array, 78

**Error Injector**, 163

Applications, 164

Modes, 166

**Exam**

Aggregate choices, 303

Attributes, 302

Concurrent Statements, 309

Control structure, 303

Counter design, 306

Design units, 297

Drivers, 309

Final, 297

Operator overloading, 306

Resolution function

Boolean, 309

Signal/variable, 304

Subprogram, 311

Types, 300

Wait, 302

**Examples**

Port association and type conversion, 23

Port association rules, 21

**F****Fields**

Std\_Logic\_Vector, 44

Testbench

disk\element\cntlrb.vhd, 49

**File**

Accessing non-consecutive elements, 113

Array

disk\subprgm\f\_array87.vhd, 116

Array emulation, 114

- Binary**
  - Type restrictions, 284
- Declaration**
  - in Procedures, 111, 113
- Multiple access**
  - disk\subprgm\file87.vhd, 114
- Simulation speed, 280**
- Binary, 281**
- Write**
  - multiple processes, 129
- Finite state machine**
  - Mealy/Moore machines, 232
  - Synthesis
  - Styles, 232
- Flip-Flop**
  - Synthesis
  - Asynchronous reset, 191
- Formal**
  - Verification, 252
- Function**
  - Conversion
    - Std\_Logic\_Vector to Enumeration, 46
  - Parameter types, 110
  - Port map, 267
  - Std\_Match
    - Numeric\_Std package, 211
  - Synthesis
    - Latch Inference, 193
  - To\_Integer
    - Numeric\_Std package, 209
- Functional**
  - Verification, 247
- G**
- Garbage Collection**
  - Access type, 106
- Generate**
  - Application example, 162
  - Configuration Declaration, 33
  - reusability, 328
  - Reuse, 328
  - Statement
    - Salient points, 9
- Generics**
  - Applications, 319
  - Static, 319
- Global**
  - Signal, 285
- Globally static,**
  - Definition, 318
  - Restrictions, 319
- Gound**
  - Port mapping, 239
- Guard**
  - Synthesis, 185
  - Signals, 11
- Guidelines**
  - Bufser port, 17
  - Component declaration, 25
  - Configuration declaration, 25, 29
  - Constants, 279
  - Deallocation
    - in Subprograms, 107
  - Don't care, 212
  - Enhancing simulation speed, 270
  - Enumerated type, 279
  - File alternatives, 280
  - Files
    - Use of, 280
  - Memories, 279
  - Processes, 278
  - Resolved signals, 272
  - Signal
    - Utilization, 271
  - Synthesis
    - Asynchronous reset, 192
    - Don't care, 212
    - Tri-State, 229
  - Variable
    - Initialization, 193
  - Type conversion, 279
- H**
- Hexadecimal**
  - Conversion, 122
- Hierarchy**
  - Signal
  - Visibility, 289
- Homogragh, 64**
  - declarations, 64

**I****Identifier**

Declaration reference, 67

**IEEE**Standards, 143  
Standards VHDL, 143  
Web site, 143**If**Priority, 210  
Synthesis, 210**Image Package**, 122Application, 126  
Example, 61**Initialization**

Synthesis:, 184

**InOut Port**, 20

Buffer Port, 17

**Integer**Range, 38  
Resolution function, 51, 54  
Two's complement, 38**Internet**

FAQ, 294

**L****Latch**

Inference, 188

**Leak**Memory  
Deallocate, 106  
ReadLine, 108**LFSR (Linear Feedback Shift register)**Application example, 257  
Example, 51  
Synthesis, 132, 136  
Verification, 249**Line (TextIO)**

Signal, 291

**Linked List**

Example, 154

**Locally static**, 318**Loop Statement**

Synthesis, 210

**LPM**

Reuse, 329

**LRM**, 64*1.1.1.2 -- Ports*, 20  
12.4.3  
Configuration  
Declaration  
Deferred Binding, 31  
12.6.1 Drivers, 83  
14.1 – Predefined attributes, 302  
3.2.1 Arra types, 71  
4.3.1.2 Drivers, 84  
4.3.2.2  
Subelement association, 230  
6.5  
Null Slice, 66  
7.2.2  
Equality, 63  
7.3.1, 44  
7.3.2.2, 76  
7.3.5, Type conversion, 39  
7.4 – Static expression, 70  
7.4.2, Static Expression, 92  
8.4.1, Side effects (Procedure), 100**M****Memory**Leak  
Deallocate, 106  
ReadLine, 108  
Model, 146  
Simulation Speed:, 284**Metalogical**

Definition, 149

**MISR (Multiple Input Signature Register)**Data reduction, 281  
instead of file, 281  
Regression  
Verification, 251

- Mod**, 40
  - disk\element\count\_ea.vhd, 41
- Model**
  - Array, timing specification, 57
  - Association VCC/GND
    - disk\synths\togndvcc.vhd, 239
  - Barrel shifter, 205
    - Behavioral
      - disk\synths\barrelnc.vhd, 205
    - Synthesis
      - disk\synths\barlsyn1.vhd, 206
      - disk\synths\barrel.vhd, 207
  - Binary File
    - Read
      - \disk\potpouri\fintb93.vhd, 283
    - Write
      - \disk\potpouri\finta93.vhd, 282
  - Bit Reversal function
    - \disk\synths\bitrevs.vhd, 240
  - Bit reversal process
    - disk\synths\bitrevp.vhd, 240
  - Bit Reverse
    - disk\subprgm\bitrflct.vhd, 119
  - Constants
    - Application of
      - disk\potpourri\const.vhd, 280
  - Conversion
    - Integer to time
      - disk\subprgm\timecnvt.vhd, 117
  - Copy file VHDL'87, 112
  - Copy file VHDL'93, 113
  - Core
    - Designs, 329
  - Count-Down
    - disk
      - element\count\_ea.vhd, 41
  - Counter, 287
  - Deferring binding, 31
  - Demultiplexer, 203
    - disk\synths\demuxc.vhd, 204
    - disk\synths\demuxp2.vhd, 205
    - disk\synths\demuxpl.vhd, 204
  - Error injector, 163
    - disk\models\errinj.vhd, 170
  - Error injector configurations
    - disk\models\ej\_cfg.vhd, 177
  - Error injector testbench
    - disk\models\ej\_tb.vhd, 176
  - Flip-flop
    - Asynchronous reset, 191
    - disk\synths\dff.vhd, 196
  - Global signal
    - \disk\potpouri\hierglob.vhd, 287
  - Integer**
    - Random package
      - disk\verif\uniform.vhd, 260
  - Linear Feedback Shift Register**
    - disk\package\lfsr4.vhd, 135
  - MC14532B**
    - Cascade, 215
    - Priority encoder, 213
  - Memory**
    - C language, 157
    - Configuration
      - disk\ramfix\_c.vhd, 158
      - disk\ramlnk\_c.vhd, 159
      - disk\rampag\_c.vhd, 158
    - Disk paging, 156
    - Dynamic page, 154
      - disk\models\ramlink.vhd, 156
    - Fixed Array, 149
      - disk\models\ramfix.vhd, 152
    - Fixed page, 152
      - disk\models\rampage.vhd, 154
    - Testbench, 157
      - disk\models\ram\_tb.vhd, 158
    - Traditional, 146
      - disk\models\memory.vhd, 149
  - Multiplexer**, 202
    - disk\synths\mux.vhd, 203
  - Multiplier**
    - disk\synths\multpy.vhd, 243
  - Pre-charge**, 220
  - Priority Encoder**, 212
  - Register file**
    - disk\synths\regfile.vhd, 200
    - Variables, 200
  - Shared variable**
    - TextIO.Line
      - \disk\potpouri\fline93.vhd, 292
  - Shift register**, 197
    - disk\synth\regok.vhd, 198
    - disk\synth\reg9.vhd, 197
    - disk\synth\regok2.vhd, 199
  - Strings to screen**
    - disk\models\strgnout.vhd, 128
  - Synthesis**
    - Counter
      - disk\synths\count.vhd, 189
    - Counter latch
      - disk\synths\countvar.vhd, 190
    - Flip-Flop
      - synths\d2ff\_asyn.vhd, 192
      - synths\dffasync.vhd, 191
      - synths\dffsync.vhd, 191
  - TextIO.Line**
    - String
      - dsik\potpouri\fline87.vhd, 293

<b>Timer</b>	<b>Numeric_Std package</b>
<b>Integer</b>	Signed/Unsigned, 60
\disk\synths\timer_ea.vhd, 241	
<b>Unsigned</b>	
\disk\synths\timr3_ea.vhd, 242	
<b>Waveform generation</b>	<b>O</b>
disk\element\wave.vhd, 8	
<b>Zero Ohm bank</b>	<b>One-Hot encoding</b>
disk\models\zohm_ea.vhd, 162	Synthesis, 233
<b>Zero Ohm resistor</b>	
disk\models\z0quiet.vhd, 161	
disk\models\zohm0_ea.vhd, 160	
<b>Modeling</b>	<b>Operator</b>
Behavioral/RTL, 296	&
	in Case statement, 42
<b>Models for Core</b>	Long strings, 43
Reuse, 329	
<b>Multiple Input Signature Register</b>	Implicit
see MISR	Array, 58
<b>Multiplexer</b>	Non-Associative, 41
Synthesis, 202	disk\element\left2r.vhd, 70
<b>Multiplier</b>	Overloading
Synthesis, 242	Equal, 63
	Synthesis (in), 184
<b>N</b>	<b>Others</b>
<b>Non-Associative Operator, 41</b>	See Array, 76
disk\element\left2r.vhd, 70	Synthesis, 76
<b>Non-Portable</b>	aggregates, 75
Deferred Binding, 31	
<b>Normalization</b>	<b>Out Port, 19</b>
<b>Array</b>	disk\element\buffer.vhd, 20
in Subprogram, 118	
<b>Notation</b>	<b>Overload</b>
Recommended, 316	<b>Operator</b>
<b>Null</b>	Explicit, 61
<b>Array</b> , 65	Equal, 63
<b>Synthesis</b>	Exam, 306
Example, 206	
<b>Numeric_Bit</b>	<b>P</b>
Source, 143	
<b>Numeric_Std</b>	<b>Package</b>
Source, 143	Accelerated - list, 285
	Conversion
	Integer to time, 116
	typed objects to string, 122
	disk\package\image_pb.vhd, 126
	<b>Image</b>
	disk\pacakge\image_pb.vhd, 126
	<b>Linear Feedback Shift Register, 132</b>
	disk\package\lfsrstd.vhd, 133
	<b>Multiple Input Shift Register, 130</b>
	<b>Multiple Input Signature Register</b>
	disk\package\misr_pb.vhd, 131
	<b>Numeric_Std</b>
	Signed/Unsigned, 60
	To_Integer function, 209

- Priority Encoder**, 212
- Random generator
  - disk\package\rndm\_int.vhd, 137
  - Integer, 137
- Std\_Lock\_Arith
  - Signed /Unsigned, 60
- Synthesis
  - Supported, 184
- Parameterization**
  - Accordion, 318
  - Alias, 320
  - Control, 319
  - Records, 320
  - Subtypes, 320
- Parameterized**
  - Priority Encoder**, 212
- Parameters**
  - Accordion/control, 316
- Path definition**, 317
- Pipeline**
  - Synthesis, 224
- Port**
  - Association list
    - disk\element\alist\_ea.vhd, 23
    - Type conversion, 22
  - Association rules
    - Open, 20
  - InOut, 20
  - InOut versus Buffer, 17
  - Out, 19
  - Source, 93
  - Synthesis
    - Initialization, 184
- Port Association**
  - Open
    - disk\element\open\_ea.vhd, 22
- Port map**
  - Function call (in), 267
- Printing**
  - from VHDL, 126
- Priority**
  - Case statement, 209
  - IF Statement, 210
- Procedure**
  - Concurrent, 193
  - File declaration, 111
  - Scope of visibility, 101
  - Side Effects, 100
- Process**
  - Clocked
    - Synthesis, 186
  - disk\element\process.vhd, 7
  - Salient points, 6
  - Sensitivity
    - Activity, 89
    - Resolved signal, 87
    - Std\_Lock\_Vector, 87
  - Variable
    - Lifetime, 193
- Pre-charge**
  - Model, 220
- Q**
- Qualifier (type)**, 44
- R**
- Random**
  - Package, 137
  - Integer, 137
  - LFSR, 51
- Range constraint**, 50
- Range definition**
  - Methods, 303
- Record**
  - Parameterization, 320
- Reference**
  - [10], 131
  - [11], 183
  - [12], 195
  - [13], 195
  - [14], 235
  - [15], 252
  - [16], 270
  - [17], 220
  - [2], xxvii, 245, 270
  - [3], xxvii
  - [4], xxvii

- [5], xxviii
- [6], 44, 48, 273
- [7], 58
- [8], 92
- [8], 245
- [9], 104
  
- Register**
  - Shadow
  - Synthesis, 224
  
- Register file**
  - Signal, 199
  - Variable, 200
  
- Register model**
  - Synthesis, 186
  
- Regression**
  - MISR/LFSR example, 257
  - Verification, 250
  
- Rem**, 40
  
- Requirements**
  - in Design Processes, 314
  
- Resolution Function**
  - Boolean
    - Exam, 309
    - Array, 86
  
- Resolved signal**
  - Activity, 85
  
- Resource sharing**
  - Synthesis, 235
  
- Reuse**
  - Attributes, 302, 328
  - BFM, 330
  - Block, 323
  - Coding styles, 316
  - Core
    - LPM, 329
  - Core models, 329
  - Design for, 313
  - Design Processes, 314
  - DesignWare, 329
  - Documentation, 331
  - Generate, 328
  - Parameters, 316
  - Scaleability, 334
  - Subprogram, 328
  
- Rollover**
  - Methods, 241
  
- S**
  
- Scaleability**
  - Reuse, 334
  
- Selected Signal Assignment**
  - Example, 7
  
- Semaphore**
  - Shared variable, 50
  
- Sensitivity**
  - Resolved composite, 87
  
- Sensitivity rule**
  - Synthesis, 186
  
- Sequential statement**
  - Synthesis, 185
  
- Shared variable**, 50, 291
  - Example
    - Line, 291
    - Semaphore, 50
  
- Shift Register**
  - Synthesis, 197
  
- Side effects**
  - Avoidance method
  - Client/server, 103
  - Concurrent procedure, 8
  - in Concurrent procedure, 8
  - in Procedure, 100
  
- Signal**
  - Access type
    - TextIO.Line, 291
  - Exam, 304
  - Global, 285
    - Hierarchy, 285
  - Hierarchy
    - Visibility, 285
  - Internal to component, 285
  - Simulation speed
    - Composite, 271
    - Implicit, 272
    - Reassignment, 271
    - Resolved, 272
  - Synthesis
    - Bus, 184

- Initialization, 184
- Register, 184
- Utilization, 271
- Signed**
  - Operations, 60
- Simulation**
  - Stopping, 281
- Simulation speed**
  - Composite assignment, 271
  - Concurrent statement
    - Disabling, 275
  - Concurrent signal assignment, 273
  - Concurrent statement, 273
  - Constant, 279
  - Enhancing methods, 270
  - File, Binary, 280, 281
  - Generate, 276
  - MISR, 281
  - Processes, 273
  - Redundant code, 277
  - Signal
    - Implicit, 272
    - Resolved, 272
  - Signal, 270
  - Signal reassignment, 271
  - Subprograms, 284
  - Types
    - Integer/Std\_Logic, 278
  - Verifier, 281
  - VITAL, 285
- Slices**
  - Null
  - Synthesis, 65
- Standards VHDL**
  - IEEE, 143
- Static**
  - Global/Local, 318
- Static expression**
  - LRM
    - 7.4.2, 92
    - 7.4, 70
  - range constraint, 50
- Std Logic**
  - VHDL'87/VHDL'93*
    - Differences, 43
    - disk\element\confused.vhd, 44
- Std\_Logic\_Arith package**
  - Signed/Unsigned, 60
- Std\_Logic\_Vector**
  - Resolution, 162
- Stopping**
  - Simulation, 281
- Strength stripper**, 267
- String**
  - Long, 43
  - See conversion, 122
- Subelement Association**
  - Application example, 173
  - in ports, 231
  - LRM 4.3.2.2.2, 230
  - Synthesis, 230
- Subprogram**
  - Exam, 311
  - Normalization of arrays, 118
  - Others, 76
  - overloading
    - LRM 2.3, 64
  - reusability, 328
  - Reuse, 328
  - See Procedure/Function, 100
  - Simulation Speed, 284
  - Variable
    - Initialization, 195
- Subtype**
  - Range definition, 303
  - Exam, 303
  - Integer
    - Parameterization, 320
- Synthesis**, 65
  - Aggregate, 185
  - Alias, 184
  - Also see model, 186
  - Architecture, 184
  - Array, 73, 186
    - Allowed, 56
  - Array
    - Memory, 57
    - Two-dimensional, 57, 199
  - Assertion, 185
  - Assignment
  - Signal, 185

Barrel shifter, 205  
Behavioral, 243  
Bit Reversal, 240  
Block  
    Guarded, 17  
    Restrictions, 185  
Block Statement, 10  
    Guarded, 17  
Buffer drivers, 225  
Bus, 184  
Case  
    & operator, 42  
    Select, 70  
Component  
    Predefined, 199  
Concurrent statements, 185  
Configurations, 185  
Constructs  
    Supported/Unsupported, 184  
Count-Down  
    disk  
        element\count\_ea.vhd, 41  
Demultiplexer, 203  
DesignWare, 234  
Documentation, 336  
Don't care, 207  
EDIF, 336  
Entity, 184  
Enumeration  
    Enum\_Encoding, 47  
File, 184  
Finite State Machine  
    Styles, 232  
Flip-flop  
    Asynchronous reset, 191  
Function  
    Latch inference, 193  
Generate statement, 9  
Guard, 185  
Guarded signals, 12  
If statement, 210  
Inequality operator, 238  
Initialization  
    Object, 184  
Integer subtype, 208  
Latch  
    Inference, 188  
Latch/register, 186  
*Latches*  
    in *Tri-states*, 227  
LFSR register, 136  
Loop statement, 210  
Memory, 199  
Model documentation, 336  
Multiplexer, 202  
Multiplier, 242  
Now, 185  
Null array  
    Example, 206  
One-Hot encoding, 233  
Operators, 184  
Optimizing for area, 238  
    Design, 236  
Package  
    Supported/Unsupported, 184  
Pipeline, 224  
Port Mapping to VCC/Ground, 239  
Register  
    Inference, 188  
Register File, 199  
Register outputs  
    Need for, 225  
Register/Combinational, 188  
Resolution function  
    User defined, 185  
Resource sharing, 235  
    Styles, 235  
Sensitivity rule, 186  
Sequential statement, 185  
Shadow register, 224  
Shift Register, 197  
Signal  
    Register, 184  
Simulation mismatch, 83  
Slice  
    Computational, 206  
    Others, 76  
Std\_Lock\_Arith  
    + integer, 189  
Subelement association, 230  
Technology impact, 223  
Timer  
    Up/Down  
        Techniques, 241  
Tri-State  
    Guideline, 229  
    Rules, 226  
Type  
    Supported, 184  
Types, 95  
Unresolved  
    Arithmetic operations, 95  
Unresolved type  
    usage, 95  
Variable, 186, 193, 197  
    Initialization, 193  
    Process, 193  
    Register, 197  
Variable Initialization  
    Subprogram, 193, 195  
Wait, 185  
    Restrictions, 195

**T**

**Technology**  
Synthesis, 223

**Testbench**

BFM  
Design, 254  
Documentation, 338  
MISR/LFSR example, 258  
Validation Plan, 248

**TextIO**

Std\_Loic  
Application, 122

**Timer**

Up/Down  
Techniques, 241

**Timing checkers**

Verification, 249

**Tri-State**

Synthesis, 226

**Two's complement**

Integer, 38

**Type**

Array (LRM 3.2.1), 71  
Exam, 300  
Conversion, 39  
diskelement\convn.vhd, 40  
Qualifier, 39

**Type conversion**

Port association lists, 22

**U****Unconstrained Array Aggregate, 75**

**Unsigned**  
Operations, 60

**V****Variable**

Exam, 304  
Initialization  
Subprogram, 195  
Synthesis, 193  
Lifetime, 193  
Process, 193

Shared, 50, 291  
Synthesis  
Initialization, 184, 193

**VCC**

Port mapping, 239

**Verification**

Automatic, 247  
Design, 248  
Design verifier  
Technique, 251  
Documentation, 337  
Error injection, 163  
File compare  
Technique, 250  
Formal, 252  
Regression, 252  
Functional, 247  
LFSR  
Application example, 257  
Technique, 249  
MISR  
Application example, 257  
Processes, 246  
Pseudo-random  
Data, 249  
Transaction, 249  
Regression, 246  
MISR, 251  
Techniques, 250  
Timing checkers, 249

**Verifier**

Data reduction, 281

**VHDL**

'98  
Requests, 272  
Item for review, 76  
VHDL'87/93 Difference, 43

**Visibility**

Scope, 101

**VITAL**

Packages, 295  
Purpose, 295  
Simulation Efficiency, 285  
Simulation Speed, 285  
Source, 143

# W

## **Wait**

- Exam, 302
- Simulator handling, 273
- Synthesis
  - Restrictions, 195
  - Synthesis:, 185
- Waveform
  - Generation example, 8
- Waveform generation
  - disk\element\wave.vhd, 8