# THE VHDL SCOUT ❁HANDBOOK❁

A Practical Introduction to IEEE Std 1076-1987,
the VHSIC Hardware Description Language

Third Edition

**COMPASS Design Automation**
**1865 Lundy Avenue**
**San Jose, CA, 95131**

DN 0529-2

Copyright © 1994 by COMPASS Design Automation, Inc.

All rights reserved. This document and the software that it describes are the property of COMPASS Design Automation, Inc. (“COMPASS”) for distribution and use only under license from COMPASS. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of COMPASS Design Automation, Inc.

COMPASS reserves the right to make changes to the contents of this document and the software without notice. COMPASS assumes no responsibility for any errors that may appear in this document or software.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013 (48 CFR 252).

Printed in U.S.A

# LICENSE AGREEMENT

PLEASE READ THE LICENSE AGREEMENT CAREFULLY BEFORE INSTALLING THIS COMPUTER SOFTWARE OR USING THE ACCOMPANYING DOCUMENTATION. COMPASS VHDL SCOUT IS COPYRIGHTED AND LICENSED (NOT SOLD). BY USING IT, YOU ARE ACCEPTING AND AGREEING TO THE TERMS OF THE LICENSE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THE TERMS OF THE LICENSE AGREEMENT, YOU SHOULD PROMPTLY RETURN THIS PACKAGE UNUSED AND YOU WILL RECEIVE A REFUND OF YOUR MONEY. THIS LICENSE AGREEMENT REPRESENTS THE ENTIRE AGREEMENT CONCERNING COMPASS VHDL SCOUT BETWEEN YOU AND COMPASS DESIGN AUTOMATION AND IT SUPERSEDES ANY PRIOR PROPOSAL, REPRESENTATION, OR UNDER-STANDING BETWEEN THE PARTIES.

## ❑ License Grant.

COMPASS Design Automation (the "Licensor") hereby grants to you, and you accept, a non-exclusive license to use this computer software and the accompanying documentation ("Scout") only as authorized in this License Agreement. Scout may be used only on a single computer owned, leased, or otherwise controlled by you. Neither concurrent use on two or more computers nor use in a local area network or other network is permitted without separate written authorization by Licensor and the payment of other license fees. You agree that you will not assign, sublicense, transfer, pledge, lease, rent, or share your rights under this License Agreement. You agree that you may not reverse assemble, reverse compile, or otherwise translate Scout.

You may make copies of Scout for backup purposes. You may make duplicate copies of the user documentation for your personal use. Any such copies shall include Licensor's copyright and other proprietary notices. Except as authorized under this paragraph, no copies of Scout or any portions thereof may be made by you or any person under your authority or control.

## ❑ Licensor's Rights.

You acknowledge and agree that Scout is a proprietary product of Licensor protected under U.S. copyright law. You further acknowledge and agree that all right, title, and interest in and to Scout, including associated intellectual property rights, are and shall remain with Licensor. This License Agreement does not convey to you an interest in or to Scout, but only a limited right of use revocable in accordance with the terms of this License Agreement.

You acknowledge and agree that Licensor can and will use technical means to enforce compliance with selected terms of this License Agreement .

## ❑ License Fees.

The license fees paid by you are paid solely in consideration of the licenses granted under this License Agreement.

## ❑ Term.

This License Agreement is effective upon your first use of Scout and shall continue until terminated. You may terminate this License Agreement at any time by returning Scout and all copies and extracts of Scout to Licensor. Licensor may terminate this License Agreement upon the breach by you of any its terms. Upon such termination by Licensor, you agree to return Scout and all copies and extracts of Scout to Licensor.

## ❑ Limited Warranty.

Licensor warrants, for your benefit alone, for a period of 90 days from the date of commencementof this License Agreement (referred to as the "Warranty Period") that the media upon which Scout has been supplied is free from defects in material and workmanship. Licensor further warrants, for your benefit alone, that during the Warranty Period Scout shall operate substantially in accordance with the functional specifications in the documentation. If during the Warranty Period, a defect in Scout appears, you may return Scout to Licensor for either replacement or, if so elected by Licensor, refund of amounts paid by you under this License Agreement. You agree that the foregoing constitutes your sole and exclusive remedy for breach by Licensor of any warranties made under this Agreement. EXCEPT FOR THE WARRANTIES SET FORTH ABOVE, SCOUT IS LICENSED "AS IS," AND LICENSOR DISCLAIMS ANY AND ALL OTHER WARRANTIES, WHETHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## ❑ Limitation of Liability.

LICENSOR'S CUMULATIVE LIABILITY TO YOU OR ANY OTHER PARTY FOR ANY LOSS OR DAMAGES RESULTING FROM ANY CLAIMS, DEMANDS, OR ACTIONS ARISING OUT OF OR RELATING TO THIS AGREEMENT SHALL NOT EXCEED THE LICENSE FEE PAID TO LICENSOR FOR THE USE OF THE PROGRAM. IN NO EVENT SHALL LICENSOR BE LIABLE FOR ANY INDIRECT, INCIDENTAL, CONSEQUENTIAL, SPECIAL, OR EXEMPLARY DAMAGES OR LOST PROFITS, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

## ❑ General Provisions.

Ths is the entire agreement between you and licensor concerning Scout. This License Agreement can only be modified by a written amendment signed by an officer of Licensor. The License Agreement shall be construed and governed in accordance with the laws of the State of California. If any action is brought by either party to this License Agreement against the other party regarding the subject matter hereof, the prevailing party shall be entitled to recover, in addition to any other relief granted, reasonable attorney fees and expenses of litigation. Should any term of this License Agreement be declared void or unenforceable by any court of competent jurisdiction, such declaration shall have no effect on the remaining terms hereof. The failure of either party to enforce any rights granted hereunder or to take action against the other party in the event of any breach hereunder shall not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent actions in the event of future breaches.

# PREFACE

*The VHDL Scout Handbook* is a hands-on, self-study course in VHDL. The core of the presentation is a series of graduated examples that illustrate how the language is used for design and simulation of electronic systems.

The *Handbook* is intended to be used with the COMPASS Scout VHDL Simulation System. The simulator software is contained on the CD-ROM bound into the back of this book.

The examples follow a bottom-up design style, building on simple concepts to develop more complex designs. We focus on the portions of VHDL that will be familiar to engineers who have used conventional design capture and digital simulation tools. Although not all of VHDL is illustrated, the examples should give a firm foundation for learning more advanced aspects of VHDL independently.

This third edition of *The VHDL Scout Handbook* (previously published under the title *Designing With VHDL*) has benefited greatly from the experience of the many readers of its predecessors over the last two years. To these too numerous to mention, we extend our thanks.

# CONTENTS

## Part 1: VHDL User's Guide


Chapter 1: Getting Started ..... 2

- o Organization of the Handbook ..... 2
- o Installing the COMPASS Scout VHDL Simulation System ..... 3
- o Why Use VHDL? ..... 4

Introduction to the Scout VHDL Simulation System ..... 5

- o Establishing a Working Library ..... 6
- o Compiling a VHDL Description ..... 7
- o Building a Simulation Model ..... 7
- o Executing a Simulation Model ..... 8
- o Examining Signal Traces ..... 8

Using VSS for Interactive Simulation ..... 11

- o Interactively Executing a Simulation Model ..... 11
- o Querying the Model ..... 12
- o Breakpoints and Stepping Commands ..... 13

o Generating and Using Command Scripts ..... 14

Chapter 2: VHDL by Example ..... 17
Modeling Equations—A Full Adder ..... 18
o Specification ..... 19
o Implementation ..... 21
o Verification ..... 23
o Running the Example ..... 25
o Additional Exercises ..... 25
Modeling Dataflow—A D-Flip Flop ..... 26
o Specification ..... 27
o Implementation ..... 29
o Verification ..... 31
o Running the Example ..... 33
o Additional Exercises ..... 33
Modeling Structure 1—A Latch ..... 34
o Specification ..... 35
o Implementation ..... 37
o Verification ..... 39

o Running the Example ..... 41
o Additional Exercises ..... 41
Modeling Structure 2—An Adder ..... 42
o Specification ..... 43
o Implementation ..... 45
o Verification ..... 47
o Running the Example ..... 49
o Additional Exercises ..... 49
Modeling Behavior—A Shift Register ..... 50
o Specification ..... 51
o Implementation ..... 53
o Verification ..... 57
o Running the Example ..... 59
o Additional Exercises ..... 61
Modeling a Finite-State Machine—A Controller ..... 62
o Specification ..... 63
o Implementation ..... 65
o Verification ..... 67

o Running the Example ..... 69
o Additional Exercises ..... 69
Packaging Declarations ..... 70
o Specification ..... 71
o Implementation—Components ..... 73
o Implementation—Utils ..... 75
o Verification ..... 79
o Running the Example ..... 81
o Additional Exercises ..... 81
Mixed Modeling—A Simple Multiplier ..... 82
o Specification ..... 83
o Implementation ..... 85
o Verification ..... 87
o Running the Example ..... 89
o Additional Exercises ..... 89
The Full Power of VHDL ..... 90

**Part 2: VHDL Reference Guide**

Chapter 1: Simulator Command Summary ..... 92

Top-Level Commands ..... 92

o Library Commands ..... 92

o Compile Command ..... 94

o Build Command ..... 94

o Simulate Command ..... 95

o Signal Trace Formatting Command ..... 95

Interactive Simulation Commands ..... 96

o Simulation Macro-Step Commands ..... 96

o Simulation Micro-Step Commands ..... 96

o Model Information Display Commands ..... 97

o Breakpoint Control Commands ..... 98

o Region Traversal Commands ..... 98

o Signal Trace Control Commands ..... 99

o Mode Control Commands ..... 100

General Commands ..... 101

o Command File Generation and Execution Commands ..... 101

o Miscellaneous Commands ..... 101

Chapter 2: VHDL Language Summary ..... 103

VHDL Lexical Elements ..... 103

o Identifiers and Reserved Words ..... 103

o Character Literals ..... 104

o String and Bit String Literals ..... 105

o Numeric Literals ..... 105

o Aggregates ..... 106

Design Units ..... 107

o Context Clause ..... 107

o Entity Declaration ..... 107

o Architecture Body ..... 108

o Package Declaration ..... 108

o Package Body ..... 108

Declarations ..... 109

o Component Declaration ..... 109

o Constant Declaration ..... 109

o Function Body ..... 110

o Function Declaration ..... 110

o Interface Declaration ..... 111
o Procedure Body ..... 111
o Procedure Declaration ..... 112
o Signal Declaration ..... 112
o Subtype Declaration ..... 112
o Type Declaration ..... 113
o Use Clause ..... 113
o Variable Declaration ..... 114
Statements ..... 114
o Assert Statement ..... 114
o Case Statement ..... 115
o Component Instantiation Statement ..... 116
o Conditional Signal Assignment Statement ..... 116
o Exit Statement ..... 117
o For Loop ..... 117
o Generate Statement ..... 117
o If Statement ..... 118
o Indefinite Loop ..... 119

o Null Statement
.....
119
o Procedure Call
.....
119
o Process Statement
.....
119
o Return Statement
.....
120
o Selected Signal Assignment Statement
.....
120
o Simple Signal Assignment Statement
.....
121
o Variable Assignment Statement
.....
121
o Wait Statement
.....
121

Chapter 3: VHDL Predefined Attributes
.....
123
Attributes of Types and Subtypes
.....
123
o Base Type
.....
123
T'Base
.....
123
o Scalar Subtype Bounds
.....
123
T'Left
.....
124
T'Right
.....
124
T'High
.....
124
T'Low
.....
124
o Position–Value Conversion
.....
124

T'Pos(X) ..... 125
T'Val(X) ..... 125
o Value Navigation ..... 125
T'Succ(X) ..... 125
T'Pred(X) ..... 125
T'LeftOf(X) ..... 125
T'RightOf(X) ..... 125
Attributes of Arrays ..... 126
o Array Index Bounds ..... 126
A'Left [(N)] ..... 126
A'Right [(N)] ..... 126
A'High [(N)] ..... 126
A'Low [(N)] ..... 126
o Array Index Ranges ..... 127
A'Range [(N)] ..... 127
A'Reverse_Range [(N)] ..... 127
A'Length [(N)] ..... 127
Attributes of Signals ..... 127

o Signal-Valued Attributes ..... 127
S'Delayed [(T)] ..... 128
S'Stable [(T)] ..... 128
S'Quiet [(T)] ..... 129
S'Transaction ..... 129
o Function-Valued Attributes ..... 129
S'Event ..... 129
S'Active ..... 129
S'Last_Event ..... 129
S'Last_Active ..... 129
S'Last_Value ..... 129
Attributes of Blocks ..... 130
o Behavioral Description Query ..... 130
B'Behavior ..... 130
o Structural Description Query ..... 130
B'Structure ..... 130

Chapter 4: VHDL Standard Packages ..... 131
Package Standard ..... 131

Package TextIO
.....
132
Package std_logic_1164
.....
135

Index
.....
151

# PART 1: VHDL USER'S GUIDE

# CHAPTER 1
# GETTING STARTED

**VHDL** is a comprehensive language for digital hardware design. It is fast becoming the worldwide standard language within the Electronic Design Automation industry. This volume provides a self-paced introduction to the concepts and constructs of IEEE Standard 1076-1987 VHDL.

Several other excellent texts are available that provide an introduction to VHDL. However, simply reading about VHDL will not teach you how to use it; practice is also essential. With that in mind, the *VHDL Scout Handbook* is built around a series of VHDL examples that illustrate the essential features of the language. We have provided the examples in machine readable form along with the VHDL Simulation System software you'll need to work through them. VHDL Scout will make it easy for you to get started with VHDL.

## ❑ Organization of the Handbook

The *VHDL Scout Handbook* is organized into two parts. Part I, VHDL Users' Guide, describes how to use the Scout VHDL Simulation System and contains the tutorial introduction to VHDL.

Each example of the tutorial starts out with a simple specification followed by a VHDL implementation of that specification. Then, a VHDL "test bench" is described; this is how many VHDL simulators use VHDL as a simulation control language. Next are directions for simulation of the example, and a discussion of the simulation results. Finally, there are suggested exercises based upon the example. We encourage you to work through the examples and the exercises in order to become fluent with VHDL.

Part II is the VHDL Reference Guide. It provides a comprehensive reference to the features and functions of the Scout VHDL Simulation System and a concise summary of the syntax, semantics and concepts of VHDL. Part II includes the full text of the standard VHDL packages you will encounter in your design work.

## ❑ Installing the COMPASS Scout VHDL Simulation System

The Scout VHDL Simulation System, the software component of COMPASS' VHDL Scout, is a medium-performance VHDL simulator suitable for VHDL training and VHDL evaluation.

Included with the Scout VHDL Simulation System are commands to compile VHDL source to object code, commands to build a simulation model from the compiled code, a suite of library management and other useful commands, and a shell environment to execute the model.

The shell environment allows the model to be executed using line-oriented commands. The simulation can be run to completion without user intervention or the user can monitor and control the simulation run as it progresses.

You will need a license key to complete installation. The license key may be supplied by mail, fax or in electronic form. In each case, it is a short text document. If you have not received a license key, contact your COMPASS representative and, following his instructions, provide the "lmhostid" for your computer. He will arrange to ship your customized license key. You will be able to complete the software installation only when you receive your license key.

Login as Super User. Determine where you want to load the software. You will need about 5 megabytes of space. Change your current default directory to that location. This directory is called "<installation_dir>" in the sample commands. Insert the CD- ROM from the back of this book into your CD-ROM drive and mount it:

```
% cd <installation_dir>
% mount -r -t hsfs /dev/sr0  <mount point>
```

The mount command will be slightly different on systems other than Sun OS 4.x; consult your local Unix expert for assistance.

Substitute an absolute directory path for "<mount point>". If the directory does not exist, you must create it.

Get a listing of the directory on the CD-ROM:

```
% ls <mount point>
```

You will see one file with the extension ".tar" or " ".TAR" for each supported platform. Select the file for your machine; we will call it "<tar archive>". Unpack that file with the following command:

```
% tar xvf <mount point>/<tar archive>
```

Now log in again, this time to the account from which you will use the Scout VHDL Simulation System.

If you use "csh" as your command shell, modify your ".cshrc" file by adding the following lines:

```
setenv CLSI <installation_dir>/clsi
source $CLSI/env/CLSISetup.csh
```

where "<installation_dir>" is the path name of the directory where you installed the software. If you use a shell other than "csh", these commands will be slightly different; consult your local Unix expert for help (an alternate set up script called "CLSISetup.sh" is provided for this purpose.)

You may prefer to define environment variables only when you log in rather than each time you start a new shell; see the "CLSISetup" script for more details.

Install the license key as the file "$CLSI/license/license.dat". You must use one of two procedures depending on how the license key was delivered to you.

If you received the license key by electronic mail, extract it from your mail system and store it as "$CLSI/license/license.dat".

If you received the key on paper, use a text editor to edit the file "$CLSI/license/sample_license.dat" so that it matches the key. Then store the result as "$CLSI/license/license.dat". Edit the file carefully. For most fields in the key, every character is significant.

Before you proceed to the next section, log out and log back in again to initialize your environment for doing the exercises.

If the Scout VHDL Simulation System does not work according to the instructions in this handbook, first ensure that you are working with unmodified copies of the examples. Each example in Chapter 2 generally depends on earlier examples; if you have modified an earlier example you may get unexpected results when a later example attempts to use the modified example. If you wish to modify an example, first make a local copy.

Another common problem occurs when a model on which another model depends is modified. If the second model is not recompiled after the recompilation of the first model, errors will result during subsequent processing steps.

When in doubt, recompile and rebuild all models from their sources.

If all else fails, reinstall the Scout VHDL Simulation System. A file may have been corrupted or accidentally modified. The system can be installed anywhere in the file system, so it is not necessary to overwrite an existing installation to determine if the existing installation has been corrupted.

## ❑ Why Use VHDL?

Designing an electronic system is a complex task. During the last decade, many software tools have been developed to support that task. The most widely used tools are schematic capture tools for drawing circuit diagrams, simulation tools for verifying the function of the design, timing analysis tools for verifying the timing of the design, and layout tools for creating the physical realization of the design. You probably use of one or more of these tools in your work today.

VHDL provides a new approach to electronic system design, one that complements existing tools and extends the capabilities of a design system to new levels. It allows you to describe a system in more abstract terms than is possible with conventional tools, and that in turn makes it possible to explore more alternatives faster and more easily than with conventional tools.

Through simulation, conceptual designs can be evaluated for both function and performance, so that only the best architectures are implemented in detail. Since VHDL also supports the lower levels of abstraction found in conventional tools, you can use VHDL from concept through detailed logic design without having to convert your design from one notation to another.

One of the primary reasons for learning VHDL is to take advantage of the VHDL synthesis tools now becoming available. Such tools automate much of the tedious process of reducing an abstract design to detailed logic, producing a more optimal implementation more quickly and with no mistakes. VHDL synthesis promises to be one of the most significant factors in cutting costs and time to market in the coming years.

## Introduction to the Scout VHDL Simulation System

Although we can reason about a VHDL description just by looking at the source code, it is much easier to understand the dynamic implications of a VHDL description by simulating it. Through simulation, we can apply the same kinds of stimulus that will be applied to the design when it is actually built, and we can determine whether it will function correctly before incurring the expense of implementation.

To illustrate the steps involved in simulating a VHDL description, we will use a very simple example reminiscent of the "hello, world" example from the Kernighan and Ritchie classic, *The C Programming Language*. However, unlike C, VHDL involves the concept of time, so our formulation of the "hello, world" example is slightly different. Instead of printing the entire string in one operation, our first VHDL description is required to print each character of the string, one after another, with a delay of 1 nanosecond (ns) between each character.

Here is the VHDL source code to do that:

```
-----------------------------------------------

entity Simple is
end Simple;

-----------------------------------------------

architecture Example of Simple is
  constant Message: String := "hello, world";
  signal Channel: Character := ' ';
begin

  P1: process
      begin
        for i in Message'Range loop
          Channel <= Message(i);
          wait for 1 ns;
        end loop;
        wait;
      end process;

end Example;
```

```
-------------------------------------------
```

The VHDL description contains a single process that will execute as follows. It starts up at the beginning of time; each time it runs, it takes one character of the string “hello, world” (the value of constant Message) and assigns that to signal Channel, and then it waits for 1 nanosecond. When the loop is finished, it will execute the final wait statement and go to sleep permanently.

The rest of this section uses this example to illustrate the steps involved in building and running a simulation. We recommend that you execute the same steps with the Scout VHDL Simulation System as you read along. Source code files for this and other examples in this book are contained in the directory $CLSI/examples/tutorial ($CLSI is defined as part of the installation of the Scout VHDL Simulation System.)

## ❑ Establishing a Working Library

The first step is establishing a working library. The working library, implemented as a directory, is where the results of compiling, building, and executing a simulation model will be stored. Enter the following command to create a new library:

```
% setenv dls_mylib /directory/path
%
```

Replace the string "mylib" with the name of your choice. The "dls_" part is required. Replace the string "/directory/path" with the path name of an existing, writable directory. You have now defined the permanent name of the library to be stored in the directory. Once you have stored design units in the library the permanent name cannot be changed.

Finally, enter the following command to establish your new library as the VHDL default library:

```
% setenv WORK dls_mylib
%
```

"WORK" is now an alternate name for the new library. Unlike a permanent name, an alternate name may be reassigned to a different library at any time. The name of the environment variable representing an alternate library name must not begin with "dls_" and must be a legal VHDL library name. The *value* of the environment variable must be a permanent library name.

You may refer to a library from within a VHDL design unit using either its permanent name (without the "dls_" part) or an alternate name. You may wish to place these two “setenv” commands in your shell start-up file so that you will not need to reenter them each time you log on.

## ❑ Compiling a VHDL Description

The next step is compiling the description. This step verifies that the syntax and static semantics of the VHDL file are correct, and it generates both an object file for simulation and a VHDL library unit (containing symbolic information required for debugging.) Both object file and library unit are stored in your VHDL design library.

First, make a local copy of the example file:

```
% cp $CLSI/examples/tutorial/simple.vhdl .
```

To compile the file, enter the following:

```
% compile simple
```

Although the full file name is `simple.vhdl`, the default extension for compilation is `.vhdl`, so you do not need to specify it. In response to this command, you should see the following output:

```
......
The following units have been compiled:
simple
simple-example

%
```

The output summarizes the results. Library units `simple` (the entity) and `simple-example` (the architecture) and their corresponding object files have been created. The library contents can be listed at any time with the *list units* command, as follows:

```
% list units
Library dls_mylib contains the following library units:

    ENTITY: simple (top level entity)
        ARCHITECTURE: example

%
```

## ❑ Building a Simulation Model

Next, we build the simulation model. This involves linking the object code produced in the compile step with the simulation kernel and runtime library that are part of the Scout VHDL Simulation System.

To build the simulation model, enter the following:

```
% build simple
```

You should see the following output in response to the above command:

```
Building complete.
To execute the simulation model, type 'simulate simple'.
%
```

The simulation model has been stored in the working library. The simulation models in the working library can be listed at any time by executing the *list models* command, as follows:

```
% list models
Library dls_mylib contains the following simulation models:

simple

%
```

Now we are ready to run the simulation.

## ❑ Executing a Simulation Model

For this first simulation run, we are going to execute the simulation without any interactive control over it.

To start simulating, enter the following:

```
% simulate simple -trace
```

In response to the above, you should see the following output:

```
%Sim-I-Ident, VHDL Simulation System Version 1.5.0.2.
%Sim-I-Elab, Elaborating.
%Sim-I-Init, Initializing.
%Sim-I-Simul, Simulating.
%Sim-I-Complete, Simulation Complete.
%
```

These messages are generated by the simulation kernel as simulation proceeds. The first identifies the version of the VHDL Simulation System. The next three messages report the beginning of elaboration, initialization, and simulation phases of execution. Finally, the last message reports that simulation has completed.

## ❑ Examining Signal Traces

After simulation is completed, you may notice that your current directory now has a file in it named `simple.trace`. This file contains a very concise record of all signal activity during simulation. In order to examine the value of a given signal over time, we need to format the particular signal's trace from the trace file for the human reader. The *format* command provides this capability.

In this case, we have only a single signal in the entire simulation run, but in the general case, there may be many, many signals involved. Formatting all of those signals at one time may be unnecessary and may simply complicate the output. To allow you to control which signals are formatted out of the set of signals that have been traced, the *format* command generates a format control file listing all of the signals available for formatting. You can edit that file to include only the signals you are interested in, and then run *format* to generate the formatted waveform file for those signals.

To generate the format control file for the current example, enter the following:

```
% format simple -preview
```

You should get the following output. After the identification message, we are told that the control file is being created. Then, the "-preview" switch causes *format* to summarize the number of signals that will be formatted and the width of the output in columns. (A minimum number of columns is required for timing information.)

```
%FMT-I-Ident, VHDL Simulation System Version 1.5.0.2.
%FMT-I-Preview, 1 signal(s) will be formatted, requiring 33 columns.
```

```
%
```

What follows is the format control file ("simple.control") itself:

```
DISPLAY 1 # /simple(example).channel
```

Again, this one is trivial, but for some simulations there may be many signals involved. If you want to inhibit the formatting of certain signals, you can do that by simply commenting out the corresponding DISPLAY command in the file (by inserting a '#' character before the command.) Running *format* with the "-preview" switch will check the now-modified format control file and give you the same summary as before, but this time taking into account any modifications to the file.

When you have decided that the desired set of signals will be formatted, enter the same command without the "-preview" switch, as follows:

```
% format simple
```

You should see the following:

```
%FMT-I-Ident, VHDL Simulation System Waveform Formatter Version
1.5.0.2.
%FMT-I-Create, Creating formatted wave file: simple.wave.
%FMT-I-Width, 1 signal(s) formatted, requiring 33 columns.
%
```

Format creates the file "simple.wave". This file contains the formatted waveform for signal "channel." All that remains is to display the file on your screen:

```
% more simple.wave
```

It should look like this:

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------
       1   : /simple(example).channel

                                1
      Time(fs) + Cycle  channel
----------------------  -------
                  0+ 0:     ' '
                  0+ 1: *   'h'
...
            1000000+ 1: *   'e'
...
            2000000+ 1: *   'l'
...
            3000000+ 1: *   'l'
...
            4000000+ 1: *   'o'
...
            5000000+ 1: *   ','
```

```
...
           6000000+ 1: *   ' '
...
           7000000+ 1: *   'w'
...
           8000000+ 1: *   'o'
...
           9000000+ 1: *   'r'
...
          10000000+ 1: *   'l'
...
          11000000+ 1: *   'd'
```

As you can see, each of the characters in the string “hello, world” appears as the value of signal “channel”, one after the other, with a delay of 1000000 femtoseconds (or 1 nanosecond.)

You may have noted that our VHDL “hello, world” description is somewhat more complicated than the corresponding C program. However, just think how complex the C program would be if it had to mimic the time-oriented behavior of the VHDL description!

# Using VSS for Interactive Simulation

In the previous section we saw how to prepare a model for simulation, and how to simulate the model by executing it to completion under the control of the simulator.

While sufficient for a simple model such as our “hello, world” example, this batch method of running to completion before examining the results is unsatisfactory for larger models. Accordingly, the Scout VHDL Simulation System also provides an interactive mode of simulation from the shell, where we can interact with the simulator as it is simulating our model.

This interactive mode has several advantages over the batch mode. We can examine results while the model is executing, so that we can avoid potentially long simulation runs where an error occurs a short way into the run. We can debug the model more effectively once an error occurs, as we can examine intermediate states in the model's execution without resorting to the introduction of extra signals. Finally, we can generate and execute scripts that automate various simulation runs.

In the remainder of this section, we use the “hello, world” model prepared in the previous section to demonstrate interactive simulation from the shell. If you have altered your environment since last simulating this example (for instance, by logging out of your computer), you should first re-establish the default library by setting the "WORK" environment variable to the permanent library name of your working library (for example, "dls_mylib".)

## ❑ Interactively Executing a Simulation Model

To start the interactive simulation of our simple model, enter the following command:

```
% simulate simple -interactive -trace
```

In response, you should see the following output:

```
%Sim-I-Ident, VHDL Simulation System Version 1.5.0.2.
Command>
```

As in batch mode, the simulator first identifies itself. Then, it displays its interactive prompt, and allows you to enter commands. A complete description of the available commands appears under Interactive Simulation Commands in Part 2. On-line help is available for each command. For example,

```
Command> help step
```

results in the following display:

```
  Execution control commands:
    elaborate      - run to end of elaboration
    initialize     - run to end of initialization
    go             - run to next breakpoint
    next           - step to next statement, w/o entering subprograms
    step           - step to next statement, entering subprograms
    step cycle     - continue to end of simulation cycle
    step process   - continue until process activation
    step suspend   - continue until process suspension
    finish         - complete execution of call
Command>
```

Usually, some commands that are not defined or used in this tutorial will be visible on the help display.

## ❑ Querying the Model

Next, we elaborate the model, so that we can display its various attributes:

```
Command> elaborate
%Sim-I-Elab, Elaborating.
Break: line = 8 of file /directory/path/simple.vhdl
        architecture Example of Simple is

Command>
```

The first attribute we will display is the model's static hierarchy:

```
Command> show hierarchy
Simulation Model Hierarchy:
/STANDARD: (Package)
/simple(example): (DesignEntity)
/simple(example).p1: (Process)
-- end of list --
Command>
```

At the top level, we see both package Standard, where VHDL's predefined environment is obtained, and the design entity "simple(example)", our model. Within simple(example), we have the process "p1".

We can also display just the processes:

```
Command> show processes
Processes:
/simple(example).p1
                  active this cycle; no breakpoint set
-- end of list --
Command>
```

and we can show the signals:

```
Command> show signals
Signal       Signal
Number       Path name (* indicates tracing on)
----------   ---------------------------------
       1   : /simple(example).channel *
-- end of list --
Command>
```

Having elaborated the model, we are now ready to begin simulation.

## ❑ Breakpoints and Stepping Commands

At this point, we could enter the *go* command, and the simulation would run to completion. However, doing so would defeat the purpose of running the simulation interactively—we wish to watch, and possibly control, the simulation as it progresses.

So instead we first set a breakpoint, which will cause the simulator to suspend its execution of the model when encountered:

```
Command> break process p1
Break set for processes p1
Command>
```

Entering the *go* command at this point then causes the simulator to execute until process P1 begins executing:

```
Command> go
%Sim-I-Init, Initializing.
Break: line = 15 of file = /directory/path/simple.vhdl
                for i in Message'Range loop

Command>
```

Since the model's signals were not previously initialized, the simulator first does so. Then, the process begins executing and the breakpoint traps the execution when the first line of the process is about to be executed, the beginning of a for loop. Control returns to the command level, where additional commands can be entered.

We can now begin executing individual lines of the process with the *step* command:

```
Command> step
Break: line = 16 of file = /directory/path/simple.vhdl
                  Channel <= Message(i);
```

Note that, in general, commands may be abbreviated to any unique prefix. However, to make single-stepping through the source code easier, a single

character 's' is always interpreted as the *step* command rather than as ambiguous, even though the *set* command also exists.

```
Command> s
Break: line = 17 of file = /directory/path/simple.vhdl
                  wait for 1 ns;

Command>
```

To return to non-interactive simulation, we must cancel all breakpoints, and then we can allow the simulation to run to completion:

```
Command> show breakpoints
No.  Breakpoint Type
---  ---------------
  1: break in process: /simple(example).p1
-- end of list --
Command> cancel break 1
Canceled break: in process: /simple(example).p1
Command> go
%Sim-I-Simul, Simulating.
%Sim-I-Complete, Simulation Complete.
Command> exit
%
```

## ❑ Generating and Using Command Scripts

It is common when simulating to execute sequences of commands repeatedly, either to initialize a model to a known state, or to execute some other function, such as the fetch-execute loop of a computer. The Scout VHDL Simulation System provides a set of commands to record and play back groups of commands to facilitate these tasks.

To illustrate this feature, consider the simple sequence of commands needed to set a breakpoint in process P1 and execute up to it. The sequence of commands is *elaborate*, *break process p1*, and *go*. We can capture these commands in a playback file by wrapping the commands *record* and *end* around the commands to be recorded:

```
% simulate simple -interactive -trace
%Sim-I-Ident, VHDL Simulation System Version 1.5.0.2.
Command> record vss.script
Recording commands to file: vss.script.
Command> elaborate
%Sim-I-Elab, Elaborating.
Break: line = 8 of file /directory/path/simple.vhdl
        architecture Example of Simple is

Command> show hierarchy
Simulation Model Hierarchy:
/STANDARD: (Package)
/simple(example): (DesignEntity)
/simple(example).p1: (Process)
-- end of list --
Command> break process p1
Break set for processes p1.
Command> go
%Sim-I-Init, Initializing.
Break: line = 15 of file = /directory/path/simple.vhdl
               for i in Message'Range loop
```

```
Command> end
Recording of commands ended.
Command> quit
%
```

We now have our commands in the file `vss.script`, the default file name for the record and playback commands:

```
% cat vss.script
#
# VHDL Simulation System Version 1.5.0.2
# Command Recording File
#
# Date: Sat Jan 1 12:02:49 1994
# User: vss_user
#
elaborate
show hierarchy
break process p1
go
```

The lines beginning with pound signs ("#") are comments. We can play these commands back with the use of the *do* command:

```
% simulate simple -interactive
%Sim-I-Ident, VHDL Simulation System Version 1.5.0.2.
Command> do vss.script
%Sim-I-Elab, Elaborating.
Break: line = 8 of file /directory/path/simple.vhdl
        architecture Example of Simple is

Simulation Model Hierarchy:
/STANDARD: (Package)
/simple(example): (DesignEntity)
/simple(example).p1: (Process)
-- end of list --
Break set for processes p1.
%Sim-I-Init, Initializing.
Break: line = 15 of file = /directory/path/simple.vhdl
                for i in Message'Range loop

Finished script: vss.script
Command>
```

We are now at the same point in this simulation as the point at which the *end* command was issued in the previous simulation.

# CHAPTER 2
# VHDL BY EXAMPLE

This chapter presents the basic concepts of VHDL through a series of nine graduated examples. The examples implement various pieces of a binary, "shift-and-add" multiplier that works in a way analogous to the way you learned to multiply multi-digit numbers in grammar school. For example, to compute 542 x 467, you actually perform 542 x 7 + 542 x 6 x 10 + 542 x 4 x 10 x 10—the multiplications by ten are implemented with decimal shifts.

In general, each example comprises four two-page panels:

- The first panel presents the functionality of the example circuit with a graphic or tabular description, as appropriate.
- The second panel presents the corresponding VHDL model of the circuit.
- The third panel presents a test bench exercising the VHDL model.
- The final panel presents the results of simulating the model and test bench with VSS; it also presents a set of additional exercises to increase your understanding of VHDL.

Within each panel, the left-hand (even-numbered) page contains the graph, table, VHDL code, or output display, while the right-hand (odd-numbered) page provides a detailed explanation.

We encourage you to simulate the examples with the Scout VHDL Simulation System as you read this chapter. All of the code is on the release media; if you follow the installation instructions exactly the examples will be in the directory `$CLSI/examples/tutorial`.

If you modify any examples, you should work with a copy of the file. Moreover, you should use a separate library for modified examples to avoid overwriting the library units used in the tutorial (more about which later.)

## Modeling Equations—A Full Adder

| X | Y | Cin | Cout | Sum |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

## ❑ Specification

Our first example is a one-bit full adder. The page opposite shows the truth table for the adder. The first three columns list the inputs to the adder, the two sum inputs, X and Y, and the carry input, Cin. Since each is a bit-level input (accepting either a '0' or a '1'), the eight rows under the header list all possible combinations of the input values.

The remaining two rows show the desired output for each input combination. There are two outputs, Sum, the single-bit result of adding X and Y, and Cout, the carry value from the addition.

Note that the sum output is the odd-parity function of the inputs. That is, whenever there are an odd number of '1's in the input (as there are in rows 2, 3, 5, and 8), the Sum output is a '1'; in the remainder of the rows there are an even number of '1's in the input and the Sum output is a '0'.

Also note that the Cout output is the majority function of the inputs. That is, Cout is '0' whenever there are zero or one '1's in the input and '1' whenever there are two or three '1's in the input.

We will make use of these properties when describing the VHDL model of the full adder.

```
 1  ------------------------------------------------
 2
 3  entity FullAdder is
 4    port (X, Y:   in      Bit;
 5          Cin:    in      Bit;
 6          Cout:   out     Bit;
 7          Sum:    out     Bit);
 8  end FullAdder;
 9
10  ------------------------------------------------
11
12  architecture Equations of FullAdder is
13  begin
14    Sum  <= X xor Y xor Cin;
15    Cout <= (X and Y) or (X and Cin) or (Y and Cin);
16  end;
17
18  ------------------------------------------------
```

## ❑ Implementation

Here is a VHDL model of the full adder. It comprises two pieces, the *entity interface*, which defines the model's interface to the outside world, and the *architecture body*, which defines the model's implementation.

The **entity interface**, shown on lines 3 through 8 on the page opposite, begins with the reserved word **entity** and then gives the entity a name, "FullAdder". (Reserved words are structural elements of VHDL and may not be used for other purposes, such as the names of things.) The entity interface continues with the reserved word **is** and ends with the reserved word **end** followed by a repeat of the entity's name and a semicolon.

Note that VHDL is a case-insensitive language in that the case of reserved words and identifiers is immaterial. "FullAdder", "fulladder", and "FULLADDER" all denote the same identifier.

Between the reserved words **is** and **end** is the entity's *port list*—see lines 4 through 7. The **port list**, beginning with the reserved word **port** and an open parenthesis and ending with a closing parenthesis and a semicolon, is a list of *interface signals* with which the model can communicate with the outside world. the port list is a semicolon-separated list of *interface signal declarations*.

Each **interface signal declaration** declares one or more interface signals. It begins with a comma-separated list of signal names (e.g., "X, Y", or "Cout") followed by a colon. Then a *mode*, here either **in** or **out**, and a *type* are specified. The type "bit" is predefined by VHDL to contain the values '0' and '1'. The mode specifies the direction of data flow between the model and the outside world with respect to the model.

Thus, the signals "X", "Y", and "Cin" can be read, but not written from within the model. When read, they pass information into the model. Similarly, the signals "Sum" and "Cout" may be written, but not read from within the model. When written, they pass information out from the model.

The **architecture body**, on lines 12 through 16 contains two *signal assignment statements* (on lines 14 and 15) that define the value of the "Sum" and "Cout" output in terms of the inputs. Each signal assignment first lists the output, then the signal assignment symbol, <=, then the expression defining the output as a function of the inputs. Here one can see that Sum's value is the xor of the three inputs (giving the odd-parity function) and Cout's value is the sum of the pairwise ands of the inputs, giving the majority function. We refer to this style as an *equational description*, since the model's outputs are defined in terms of their abstract relationship to the inputs.

One interesting characteristic of architecture bodies is that statements within them execute concurrently, or in parallel. Thus the order of the statements does not influence the model's behavior.

```
19  -------------------------------------------------
20
21  entity Test_FullAdder is end;
22
23  -------------------------------------------------
24
25  architecture Driver of Test_FullAdder is
26
27    component FullAdder
28      port (X, Y: in      Bit;
29            Cin:  in      Bit;
30            Cout: out     Bit;
31            Sum:  out     Bit);
32    end component;
33
34    signal X, Y, Cin, Cout, Sum: Bit;
35
36  begin
37
38    UUT: FullAdder port map (X, Y, Cin, Cout, Sum);
39
40    Stimulus:
41      process
42          type Entry is record
43            X,Y,Cin:      Bit;  -- input values
44            Cout, Sum:    Bit;  -- output values
45          end record;
46          type Table is array (0 to 7) of Entry;
47          constant TruthTable: Table :=
48            (--- X -- Y -- Cin ----- Cout -- Sum ---
49               ('0', '0',  '0',       '0',   '0'),
50               ('0', '0',  '1',       '0',   '1'),
51               ('0', '1',  '0',       '0',   '1'),
52               ('0', '1',  '1',       '1',   '0'),
53               ('1', '0',  '0',       '0',   '1'),
54               ('1', '0',  '1',       '1',   '0'),
55               ('1', '1',  '0',       '1',   '0'),
56               ('1', '1',  '1',       '1',   '1')
57            );
58      begin
59          for i in TruthTable'range loop
60            X    <= TruthTable(i).X;
61            Y    <= TruthTable(i).Y;
62            Cin  <= TruthTable(i).Cin;
63            wait for 1 ns;
64            assert
65              Cout = TruthTable(i).Cout and
66              Sum  = TruthTable(i).Sum;
67          end loop;
68          wait;  -- to terminate simulation
69      end process;
70
71  end;
72
73
74  -------------------------------------------------
```

## ❑ Verification

Having described a VHDL model for the full adder, we need a "test bench" that will exercise the model and examine its responses. On the page opposite is a VHDL test bench for the full adder. Like the model, the test bench also comprises an entity interface and an architecture body.

The entity interface of the test bench (line 21) has no ports as it does not need to communicate with the outside world; the test bench is self-contained. (The entity interface must be provided, even when it describes a self-contained test bench.)

The architecture body (lines 25 through 71), contains the bulk of the test bench description. It consists of the unit under test (line 38) and the tester (lines 40 through 69.)

The unit under test is modeled as a *component instance*, which is VHDL's way of creating structure. A component, "FullAdder", whose interface matches the model's, is instantiated and "wired up" to the tester. The component is declared on lines 27 through 32, and the internal signals with which its instance is wired are declared on line 34. The *port map* of the component instance connects the ports of the component to the internal signals.

The tester is modeled as a *process statement*. Within the process statement is a declaration of a table (lines 47 through 57), that describes the stimulus to be applied and the expected responses—compare this table to the function table on page 16. The type of the table is described as an array of entries on line 46, and the entries themselves are described as a collection of bits on lines 43 through 45.

The tester applies these stimuli and checks the responses in the statement part of the process, lines 59 through 68. (Within a process are *sequential statements* that execute, one after the other, in normal, programming-language fashion.) The *for loop* (lines 59 through 67, successively apply each row of the table to the inputs and check the outputs. (Note that it is not necessary to declare the loop iterator, "i"—the loop does this for you automatically.) The signal assignment statements on lines 60 through 62 apply the current input to the appropriate internal signal (which, in turn, is connected to the appropriate input of the unit under test). The *wait statement* on line 63 waits for 1 ns (nanosecond) of simulated time to elapse, giving the full adder time to calculate its outputs. The *assert statement* (lines 64 through 66) checks that the condition following the reserved word **assert** is met; in this case, the current outputs are checked against the expected responses.

When the table has been exhausted, the for loop terminates and the wait statement on line 68 is executed. Since it has no *timeout clause* (the reserved word **for** followed by a non-negative time), it waits forever, causing the tester to become quiescent. Since there is no other activity in the test bench, the simulation terminates.

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
       1   : /test_fulladder(driver).x
       2   : /test_fulladder(driver).y
       3   : /test_fulladder(driver).cin
       4   : /test_fulladder(driver).cout
       5   : /test_fulladder(driver).sum

                           1    2    3    4    5
      Time(fs) + Cycle     x    y  cin cout  sum
----------------------  ---- ---- ---- ---- ----
                  0+ 0:  '0'  '0'  '0'  '0'  '0'
                  0+ 1: *'0' *'0' *'0' *'0' *'0'
...
            1000000+ 1: *'0' *'0' *'1'  '0'  '0'
            1000000+ 2:  '0'  '0'  '1' *'0' *'1'
...
            2000000+ 1: *'0' *'1' *'0'  '0'  '1'
            2000000+ 2:  '0'  '1'  '0' *'0' *'1'
...
            3000000+ 1: *'0' *'1' *'1'  '0'  '1'
            3000000+ 2:  '0'  '1'  '1' *'1' *'0'
...
            4000000+ 1: *'1' *'0' *'0'  '1'  '0'
            4000000+ 2:  '1'  '0'  '0' *'0' *'1'
...
            5000000+ 1: *'1' *'0' *'1'  '0'  '1'
            5000000+ 2:  '1'  '0'  '1' *'1' *'0'
...
            6000000+ 1: *'1' *'1' *'0'  '1'  '0'
            6000000+ 2:  '1'  '1'  '0' *'1' *'0'
...
            7000000+ 1: *'1' *'1' *'1'  '1'  '0'
            7000000+ 2:  '1'  '1'  '1' *'1' *'1'
```

## ❑ Running the Example

To run the example and display the results from a copy in your directory, execute the following commands. Note that only the commands, and not their responses, are shown for brevity.

```
% cp $CLSI/examples/tutorial/fulladder.vhdl .
% compile fulladder
% build test_fulladder
% simulate test_fulladder -trace
% format test_fulladder -preview
% format test_fulladder
% page test_fulladder.wave
```

At left is a copy of the `wave` file you have just produced. The first section describes the signal map, the mapping from the *hierarchical name* of each signal in the test bench and model to a unique number. Each hierarchical name begins with a slash; the name of the top-level entity and architecture follows (this is always the test bench). The architecture name is surrounded by parentheses. The period indicates that the remainder of the name is interpreted in the context of the test bench; thus, the name `/test_fulladder(driver).x` is the signal "X" inside of the test bench.

The final section of the file lists the values of these signals over time. The first column gives the simulated time; the remaining columns present the value of the signals at that time. An asterisk preceding a signal value indicates that an assignment to that signal (known as a *transaction*) has taken place at the current time. Note that not every transaction results in a change to the value of the signal—those that do are called *events*.

Each pair of lines represents the application of one set of stimuli (on the first line of the pair) followed by the model's response (on the second line of the pair). Compare this output with the truth table and with the test bench's stimulus-response table.

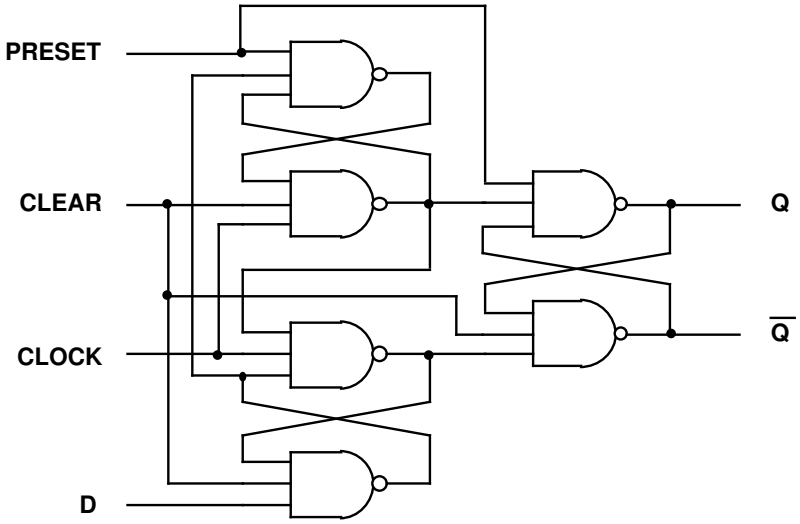
## ❑ Additional Exercises

1. Switch the order of the signal assignments in the full adder's architecture. Resimulate and compare the output of the modified version with the original. Does the order of these statements matter?

2. Edit the test bench to reflect incorrect responses to the applied stimuli. Resimulate and note the response of the assert statement in the test bench.

3. Starting with the original test bench, move the wait statement on line 63 to different places. Resimulate. Does the order of these statements matter?

## Modeling Dataflow—A D-Flip Flop



## ❑ Specification

The next example is of a clocked, D flip-flop with asynchronous, active-low preset and clear lines.

The page opposite presents a gate-level schematic of the flip-flop, which will be implemented as a VHDL model on the next page.

```
 1  -------------------------------------------------
 2
 3  entity DFF is
 4    port (Preset: in      Bit;
 5          Clear:  in      Bit;
 6          Clock:  in      Bit;
 7          Data:   in      Bit;
 8          Q:      out     Bit;
 9          QBar:   out     Bit);
10  end DFF;
11
12  -------------------------------------------------
13
14  architecture Dataflow of DFF is
15
16    signal A, B, C, D: Bit;       -- internal nodes
17    signal QInt, QBarInt: Bit;    -- internal feedback
18
19  begin
20
21    A <= not (Preset and D and B)          after 1 ns;
22    B <= not (A and Clear and Clock)       after 1 ns;
23    C <= not (B and Clock and D)           after 1 ns;
24    D <= not (C and Clear and Data)        after 1 ns;
25
26    QInt <= not (Preset and B and QBarInt) after 1 ns;
27    QBarInt <= not (QInt and Clear and C)  after 1 ns;
28
29    Q <= QInt;
30    QBar <= QBarInt;
31
32  end;
33
34  -------------------------------------------------
```

## ❑ Implementation

The page opposite presents a VHDL model of the D flip-flop. Like the "equational" representation of the full adder, we here use signal assignment statements (on lines 21 through 24, 26, 27, 29, and 30) to model the behavior of the flip-flop. However, unlike the full adder, here there is a one-to-one correspondence between the signal assignments and actual gates; moreover, the signals driven by the statements represent the actual data flow of the flip-flop. Hence, this description style is called *dataflow description*.

The signal declaration on line 16 defines the internal signals of the flip-flop, used to wire up the gates. The signals declared on line 17 serve a somewhat different purpose. Note that the interface signals "Q" (line 8) and "QBar" (line 9) are declared with mode **out**, implying that they may be written to, but not read from. Examination of the schematic will reveal that these signals participate in feedback loops, which imply that they must both be read and written. The solution here is to introduce these internal signals, then drive the output signals with the values of these internal signals—see lines 29 and 30.

Notice also that most the signal assignments have a time associated with them—all but the final two have the *after clause* `after 1 ns`. This delay is interpreted both as the propagation delay of the gate (the time delay between the arrival of an edge at an input and the appearance of a change at the output) and also as the inertial delay of the gate (the width of the shortest pulse that will propagate through the gate). Thus, considering line 21, if an edge on the "Preset" line arrives at 42 ns, "A" will not change until 43 ns.

The signal assignment statements on lines 29 and 30 do not have any delay associated with them. The lack of an after clause implies the implicit after clause `after 0 ns`. However, this implicit (or, for that matter, an identical, explicit after clause) does not mean that the outputs change instantly when an input changes. *All signal assignments have some delay associated with them.* Specifying (either implicitly or explicitly) a delay of 0 ns implies that there is an infinitesimal, or *delta*, delay associated with the signal assignment.

Look back to the simulation results on page 22. Notice that time is specified as "*n* + *m*", where *n* is a time (here measured in fs, or femtoseconds), and *m* is a dimensionless number. This number, *m*, represents the number of delta cycles at the current "real time", *n*. Thus, an event occurring at 100 ps + 4 is interpreted as taking place at some time after an event 100 ps + 3; however, the time interval between them is assumed to be so short as to not be of interest in the model.

It is a characteristic of delta delay that any number of delta cycles occurring successively at a given real time do not advance real time. Thus, time in VHDL can be thought to be two-dimensional, with the first dimension being real time, and the second being delta intervals. Each advance in real time sets the current number of accumulated delta intervals back to zero.

```
35  ------------------------------------------------
36
37  entity Test_DFF is end;
38
39  ------------------------------------------------
40
41  architecture Driver of Test_DFF is
42
43    component DFF
44      port (Preset, Clear, Clock, Data:   in      Bit;
45            Q, QBar:                      out     Bit);
46    end component;
47
48    signal Preset, Clear: Bit := '1';
49    signal Clock, Data, Q, QBar: Bit;
50
51  begin
52
53    UUT: DFF port map (Preset, Clear, Clock, Data, Q, QBar);
54
55    Stimulus:
56        process
57        begin
58
59          -- Check Preset and Clear
60
61          Preset <= '0';  wait for 5 ns;
62          Preset <= '1';  wait for 5 ns;
63          Clear  <= '0';  wait for 5 ns;
64          Clear  <= '1';  wait for 5 ns;
65
66          -- Check interaction of Preset and Clear
67
68          Preset <= '0';  Clear  <= '0';  wait for 5 ns;
69          Preset <= '1';  Clear  <= '1';  wait for 5 ns;
70
71          -- Clear it again
72
73          Clear  <= '0', '1' after 5 ns;  wait for 10 ns;
74
75          -- Check the data and clock inputs
76
77          Data   <= '1'; Clock  <= '0' after 1 ns, '1' after 5 ns;
78          wait for 10 ns;
79          Data   <= '0'; Clock  <= '0' after 1 ns, '1' after 5 ns;
80          wait for 10 ns;
81
82          -- Clear it again
83
84          Clear  <= '0', '1' after 5 ns;  wait for 10 ns;
85
86          -- Check interaction of Preset and Clock
87
88          Data   <= '0';
89          Preset <= '0', '1' after 10 ns;
90          Clock  <= '0', '1' after 5 ns;  wait for 10 ns;
91
92          -- Stop the simulation
93
94          wait;
95
96        end process;
97
98  end;
99
100 ------------------------------------------------
```

## ❑ Verification

The test bench for the flip-flop is a stimulus-generator only; it is up to the modeler to determine the correctness of the response. Rather than encoding the stimuli (and the expected responses) in a table, we have written them out in linear fashion. This stylistic difference comes about because the tests for a sequential circuit (such as a D flip-flop), especially the timing requirements, are much more complex than for a combinatorial circuit (such as a full adder).

Moreover, we are using this test bench to characterize the behavior of the flip-flop. Since the circuit is modeled as interconnected NAND gates, we cannot "read out" of it such characteristics as minimum clock period, preset and clear pulse widths, and data setup and hold times. Hence, such characteristics must be derived; the test bench is an aid to such derivations.

A new VHDL feature is introduced in the signal declaration on line 48. Notice the phrase ":= '1'" between the type (Bit) and the terminal semicolon. This phrase indicates that the declared signals (Preset and Clear) are *explicitly initialized* with the value '1'.

Another new feature is shown in the signal assignment to "Clock" on line 77. Its right-hand side comprises *multiple waveform elements*, a convenient way of providing complex waveforms to a signal. (The entire right-hand side of a signal assignment statement is called the *waveform*, while each portion of the form "*expression* **after** *time*" is a *waveform element*. The elements of a waveform are separated with commas.) We can rewrite the signal assignment to "Clock" without the use of multiple waveforms, but the resulting code is much more complex:

```
Clock <= '0' after 1 ns;
wait for 1 ns;
Clock <= '1' after 4 ns;
```

Note that we cannot merely write:

```
Clock <= '0' after 1 ns;
Clock <= '1' after 5 ns;
```

At the time of the execution of the second statement, recall that the first waveform has not acted upon Clock. When the second statement executes, recall that the 5 ns, in addition to representing propagation delay, also represents inertial delay. The '0' at 1 ns would therefore never appear in the output, since VHDL would interpret it as a spike of less than 5 ns duration.

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
       1   : /test_dff(driver).preset
       2   : /test_dff(driver).clear
       3   : /test_dff(driver).clock
       4   : /test_dff(driver).data
       5   : /test_dff(driver).q
       6   : /test_dff(driver).qbar

                              1     2     3    4    5    6
      Time(fs) + Cycle  preset clear clock data    q qbar
----------------------  ------ ----- ----- ---- ---- ----
                  0+ 0:    '1'   '1'   '0'  '0'  '0'  '0'
                  0+ 1: *  '0'   '1'   '0'  '0' *'0' *'0'
...
            1000000+ 1:    '0'   '1'   '0'  '0' *'1' *'1'
...
            2000000+ 1:    '0'   '1'   '0'  '0'  '1' *'0'
...
            5000000+ 1: *  '1'   '1'   '0'  '0'  '1'  '0'
...
           10000000+ 1:    '1' * '0'   '0'  '0'  '1'  '0'
...
           11000000+ 1:    '1'   '0'   '0'  '0'  '1' *'1'
...
           12000000+ 1:    '1'   '0'   '0'  '0' *'0'  '1'
...
           15000000+ 1:    '1' * '1'   '0'  '0'  '0'  '1'
...
           20000000+ 1: *  '0' * '0'   '0'  '0'  '0'  '1'
...
           21000000+ 1:    '0'   '0'   '0'  '0' *'1'  '1'
...
           25000000+ 1: *  '1' * '1'   '0'  '0'  '1'  '1'
...
           26000000+ 1:    '1'   '1'   '0'  '0' *'0' *'0'
...
           27000000+ 1:    '1'   '1'   '0'  '0' *'1' *'1'
...
           28000000+ 1:    '1'   '1'   '0'  '0' *'0' *'0'
...
           29000000+ 1:    '1'   '1'   '0'  '0' *'1' *'1'
...
           30000000+ 1:    '1' * '0'   '0'  '0' *'0' *'0'
...
           31000000+ 1:    '1'   '0'   '0'  '0' *'1' *'1'
...
           32000000+ 1:    '1'   '0'   '0'  '0' *'0'  '1'
...
           35000000+ 0:    '1' * '1'   '0'  '0'  '0'  '1'
...
           40000000+ 1:    '1'   '1'   '0' *'1'  '0'  '1'
           41000000+ 0:    '1'   '1' * '0'  '1'  '0'  '1'
...
           45000000+ 0:    '1'   '1' * '1'  '1'  '0'  '1'
...
           47000000+ 1:    '1'   '1'   '1'  '1' *'1'  '1'
...
           48000000+ 1:    '1'   '1'   '1'  '1'  '1' *'0'
...
           50000000+ 1:    '1'   '1'   '1' *'0'  '1'  '0'
           51000000+ 0:    '1'   '1' * '0'  '0'  '1'  '0'
```

## ❑ Running the Example

The D flip-flop has some internal signals that may not be of interest to us. In order to limit the output when running this example, we will edit the format control file to comment out (with a leading "#") the DISPLAY entries for signals 7 through 12 (the signals internal to the flip-flop). Use your favorite text editor to edit the file `test_dff.control` after the first time you run *format*. The entire sequence of commands to run this example is as follows:

```
% cp $CLSI/examples/tutorial/dff.vhdl .
% compile dff
% build test_dff
% simulate test_dff -trace
% format test_dff -preview
% # edit "test_dff.control", commenting out signals 7-12
% format test_dff
% more test_dff.wave
```

At left is a copy of the `wave` file you have just produced. (It has been truncated to fit on a single page of this manual.) Compare the output with the stimuli supplied with the test bench and satisfy yourself that the model implements a D flip-flop.

## ❑ Additional Exercises

1. The test bench currently checks the interaction between Preset and Clock (lines 86 through 90), but it does not check the interaction between Clear and Clock. Add a check to the test bench to check the interaction between Clear and Clock. Don't forget to clear the flip-flop first. Resimulate.

2. Add response checking to the test bench. Resimulate.

3. Modify the times in the test bench to determine:

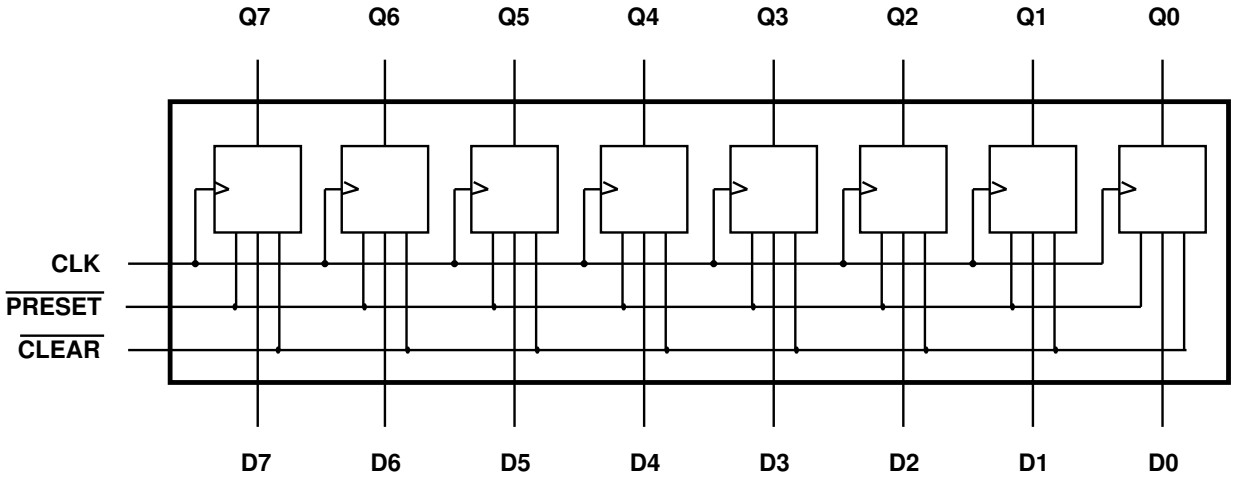
    a. the minimum clock period,
    b. the minimum pulse width on preset,
    c. the minimum pulse width on clear,
    d. the minimum setup time on data, and
    e. the minimum hold time on data.

    Resimulate.

4. Currently, when Preset and Clear are both active, Q and QBar are both '1' (see the output at time 22000000 fs + 0 on the page opposite, for example). Modify the D flip-flop model to ensure that QBar always follows Q, but inverted. Does Preset now override Clear, or does Clear override Preset?

5. If you modified your model in Exercise 4 so that Preset overrides Clear, modify your model so that Clear overrides Preset. Otherwise, modify your model so that Preset overrides Clear. Resimulate to verify.

## Modeling Structure 1—A Latch



## ❑ Specification

We now incorporate the D flip-flop into an eight-bit latch.

The page opposite presents a block diagram of the latch, showing the eight, interconnected D flip-flops. The VHDL model on the next page implements this block diagram directly.

```
 1  -------------------------------------------------
 2
 3  entity Latch8 is
 4    port (D:      in      Bit_Vector(7 downto 0);
 5          Clk:    in      Bit;
 6          Pre:    in      Bit;
 7          Clr:    in      Bit;
 8          Q:      out     Bit_Vector(7 downto 0));
 9  end Latch8;
10
11  -------------------------------------------------
12
13  architecture Structure of Latch8 is
14
15    component DFF
16      port (Preset:       in      Bit;
17            Clear:        in      Bit;
18            Clock:        in      Bit;
19            Data:         in      Bit;
20            Q:            out     Bit;
21            QBar:         out     Bit);
22    end component;
23
24    signal QBar: Bit_Vector(7 downto 0);
25
26  begin
27
28    F7: DFF port map (Pre, Clr, Clk, D(7), Q(7), QBar(7));
29    F6: DFF port map (Pre, Clr, Clk, D(6), Q(6), QBar(6));
30    F5: DFF port map (Pre, Clr, Clk, D(5), Q(5), QBar(5));
31    F4: DFF port map (Pre, Clr, Clk, D(4), Q(4), QBar(4));
32    F3: DFF port map (Pre, Clr, Clk, D(3), Q(3), QBar(3));
33    F2: DFF port map (Pre, Clr, Clk, D(2), Q(2), QBar(2));
34    F1: DFF port map (Pre, Clr, Clk, D(1), Q(1), QBar(1));
35    F0: DFF port map (Pre, Clr, Clk, D(0), Q(0), QBar(0));
36
37  end;
38
39  -------------------------------------------------
```

## ❑ Implementation

Here is a *structural* description of the eight-bit latch, so-called because the model's architecture is described entirely in terms of interconnected component instances. Let us examine some new VHDL features in this model.

Notice the "D" and "Q" signals (lines 4 and 8). They are of the predefined type "Bit_Vector", which is a one-dimensional array of bits—this is also true of the internal signal QBar, declared on line 24. The index of bit vectors runs between any two non-negative numbers, either in descending order—as we have here—or in ascending order (in which case the index range would be given as "*n* **to** *m*). The left-most number represents the index of the left-most element of the array; hence the order of the individual elements of "D" are: D(7), D(6), …, D(1), D(0).

Within the architecture, we again declare the component DFF (on lines 15 through 22), whose interface matches that of the DFF model (see page 26, lines 3 through 10). We then create eight instances of the DFF with the eight component instantiation statements on lines 28 through 35. Notice that the latch's interface signals "Pre", "Clr", and "Clk" are respectively wired to each instance's "Preset", "Clear", and "Clock" ports; however, the individual bits of "D", "Q", and the internal signal "QBar" are wired to individual ports of the flip-flops, as shown on the schematic.

It is important to note that each instance of the flip-flop (and, indeed, each instance of *any* component) are logically distinct from any other instance. In particular, events within one instance imply nothing about events in other instances.

The process whereby VHDL creates these multiple, distinct copies of a component is called *elaboration*. The elaboration of a component instance first involves determining which entity-architecture pair is *bound* to that instance. Then, a copy of that entity-architecture pair is created and "plugged in" to the instance. Finally, any particularizations with respect to the instance (such as the specific signals connected to the instance) are applied.

The binding of an entity-architecture pair to a component instance may be under the explicit control of the user, or (as we have used and will continue to use throughout this manual) it occurs subject to a set of *default binding rules*.

According to the default binding rules of VHDL, an entity with the following characteristics is selected to be bound to a given component instance:

1. The entity must have the same name as the component's name.

2. For each of the component's ports, there must be a port of the entity with the same name, mode (direction), and type as those respective properties of the component's port.

Furthermore, the default binding rules dictate that the most recently compiled architecture corresponding to the selected entity should be used. It is an error if there is no such architecture.

```
40  ------------------------------------------------
41
42  entity Test_Latch8 is end;
43
44  ------------------------------------------------
45
46  architecture Driver of Test_Latch8 is
47
48    component Latch8
49      port (D:    in      Bit_Vector(7 downto 0);
50            Clk:  in      Bit;
51            Pre:  in      Bit;
52            Clr:  in      Bit;
53            Q:    out     Bit_Vector(7 downto 0));
54    end component;
55
56    signal D, Q: Bit_Vector(7 downto 0);
57    signal Clk, Pre, Clr: Bit := '1';
58
59  begin
60
61    UUT: Latch8 port map (D, Clk, Pre, Clr, Q);
62
63    Stimulus:
64      process
65          variable Temp: Bit_Vector(7 downto 0);
66      begin
67
68          -- Set the latch
69
70          Pre <= '0', '1' after 5 ns;
71          wait for 10 ns;
72
73          -- Clear the latch
74
75          Clr <= '0', '1' after 5 ns;
76          wait for 10 ns;
77
78          -- Load the latch
79
80          Temp := "00010011";
81          for i in 1 to 8 loop
82            D <= Temp;
83            Clk <= '0' after 1 ns, '1' after 5 ns;
84            wait for 10 ns;
85            assert Q = Temp report "Load Failed";
86            Temp := Temp(0) & Temp(7 downto 1);
87          end loop;
88
89          wait; -- to terminate simulation
90
91      end process;
92
93  end;
94
95  ------------------------------------------------
```

## ❑ Verification

The page opposite presents a test bench for the latch. After setting and clearing the latch (for which no response checking is provided), it loads the pattern "00010011" into the latch and verifies that this value is asserted on the output. It then rotates this pattern right by one bit, reloads, and reverifies. This sequence of shift, load, and verify is repeated six more times until all circular permutations of this pattern have been tested.

There are several new VHDL features introduced in this test bench. The first is the *variable declaration* on line 65. Variables are storage elements local to processes; like variables in programming languages, assignments to them take place immediately. (Contrast this behavior with that of signal assignment—the signal assignment statements

```
A <= B;
B <= A;
```

swaps the values of A and B, while the variable assignment statements

```
A := B;
B := A;
```

cause the value of B to be placed in A. The use of the ":=" symbol in variable assignment and the "<=" symbol in signal assignment is intended to remind you of this difference.)

This variable is first assigned to in line 80. Here, the variable is being loaded with the pattern mentioned above as a constant, bit vector value. Note that there are eight bits in this pattern, corresponding to the number of bits in "Temp."

The assert statement on line 85 has a new clause, the *report clause*, the reserved word **report** followed by a string. Whenever the assert is executed, the report string is displayed by VSS if the assert condition is false.

Line 86 is where the right circular shift takes place. Notice the right-hand side of this variable assignment. It is the expression "Temp(0) & Temp(7 downto 1)". The "&" operator, the *concatenation* operator, is defined for one-dimensional arrays of any type (recall that Bit_Vector is a one-dimensional array of Bits). Here, the left operand, "Temp(0)" (an *index name*), yields the right-most bit of Temp. The right operand, "Temp(7 downto 1)" (a *slice name*), yields the left-most seven bits of Temp. The concatenation operator generates a result whose length is the sum of the operand lengths (here eight) and whose elements consist of the elements of the left operand, in order (from left to right), followed by the elements of the right operand. Thus, this expression circulates "Temp" by one bit to the right; this circulated value is then stored back in Temp.

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
   9 - 16  : /test_latch8(driver).q

                           9   10   11   12   13   14   15   16
      Time(fs) + Cycle  q(7) q(6) q(5) q(4) q(3) q(2) q(1) q(0)
----------------------  ---- ---- ---- ---- ---- ---- ---- ----
                  0+ 0:  '0'  '0'  '0'  '0'  '0'  '0'  '0'  '0'
                  0+ 1: *'0' *'0' *'0' *'0' *'0' *'0' *'0' *'0'
...
            1000000+ 1: *'1' *'1' *'1' *'1' *'1' *'1' *'1' *'1'
...
           12000000+ 1: *'0' *'0' *'0' *'0' *'0' *'0' *'0' *'0'
...
           27000000+ 1:  '0'  '0'  '0' *'1'  '0'  '0' *'1' *'1'
...
           37000000+ 1: *'1'  '0'  '0'  '1' *'1'  '0'  '1'  '1'
...
           38000000+ 1:  '1'  '0'  '0' *'0'  '1'  '0' *'0'  '1'
...
           47000000+ 1:  '1' *'1'  '0'  '0'  '1' *'1'  '0'  '1'
...
           48000000+ 1:  '1'  '1'  '0'  '0' *'0'  '1'  '0' *'0'
...
           57000000+ 1:  '1'  '1' *'1'  '0'  '0'  '1' *'1'  '0'
...
           58000000+ 1: *'0'  '1'  '1'  '0'  '0' *'0'  '1'  '0'
...
           67000000+ 1:  '0'  '1'  '1' *'1'  '0'  '0'  '1' *'1'
...
           68000000+ 1:  '0' *'0'  '1'  '1'  '0'  '0' *'0'  '1'
...
           77000000+ 1: *'1'  '0'  '1'  '1' *'1'  '0'  '0'  '1'
...
           78000000+ 1:  '1'  '0' *'0'  '1'  '1'  '0'  '0' *'0'
...
           87000000+ 1:  '1' *'1'  '0'  '1'  '1' *'1'  '0'  '0'
...
           88000000+ 1: *'0'  '1'  '0' *'0'  '1'  '1'  '0'  '0'
...
           97000000+ 1:  '0'  '1' *'1'  '0'  '1'  '1' *'1'  '0'
...
           98000000+ 1:  '0' *'0'  '1'  '0' *'0'  '1'  '1'  '0'
```

## ❑ Running the Example

The latch has many internal signals; each instance of the D flip-flop needs a separate copy of the signals "A", "B", "C", "D", "QInt", and "QBarInt." In order to limit the output when running this example, we will edit the format control file to comment out the DISPLAY entries for all signals except signals 9 through 16 (the output signals of the latch). Using your text editor, edit the file `test_latch.control` after the first time you run *format*. The entire sequence of commands to run this example is as follows:

```
% cp $CLSI/examples/tutorial/latch8.vhdl .
% compile latch8
% build test_latch8
% simulate test_latch8 -trace
% format test_latch8 -preview
% # edit "test_latch8.control", commenting out all signals except 9-
16
% format test_latch8 -preview
% format test_latch8
% more test_latch8.wave
```

At left is a copy of the `wave` file you have just produced. Verify the correct operation of the latch.

## ❑ Additional Exercises

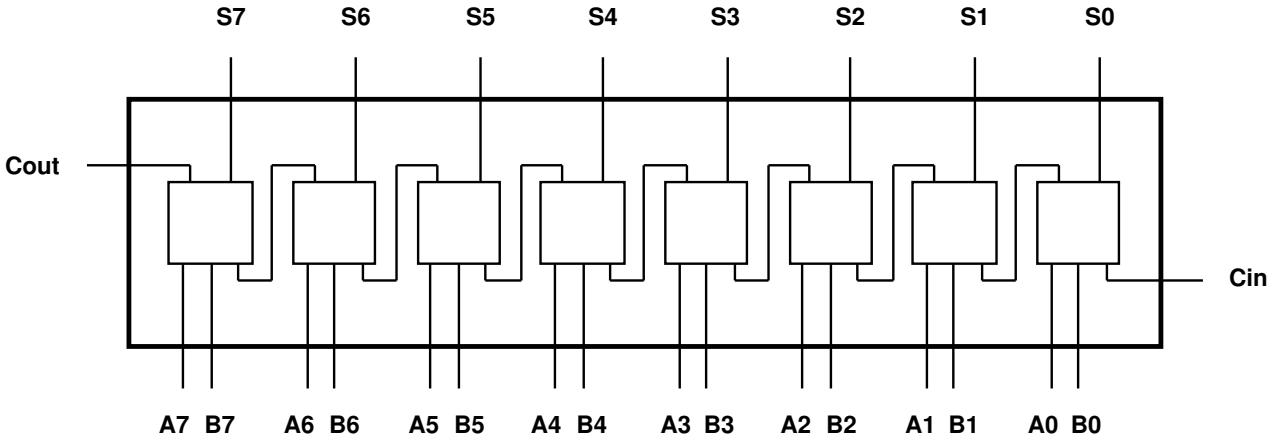
1. Modify the control file to display both the input and output signals to and from the latch.

2. Add response checking for the set and clear stimuli to the test bench. Resimulate.

3. Take the minimum clock period, minimum preset and clear pulse width, and data setup and hold times from the previous model's exercises and verify that these times are unchanged when the flip-flop is incorporated into the latch.

4. Build a 32-bit latch.

## Modeling Structure 2—An Adder



## ❑ Specification

We will now use structural description to build an eight-bit, ripple-carry adder based on the full adder.

In doing so, we will introduce a new VHDL capability that will allow one to replicate statements, easing the description of regular structures, such as latches and adders.

```
 1  ------------------------------------------------
 2
 3  entity Adder8 is
 4    port (A, B:   in      Bit_Vector(7 downto 0);
 5          Cin:    in      Bit;
 6          Cout:   out     Bit;
 7          Sum:    out     Bit_Vector(7 downto 0));
 8  end Adder8;
 9
10  ------------------------------------------------
11
12  architecture Structure of Adder8 is
13
14    component FullAdder
15      port (X, Y: in      Bit;
16            Cin:  in      Bit;
17            Cout: out     Bit;
18            Sum:  out     Bit);
19    end component;
20
21    signal C: Bit_Vector(7 downto 0);
22
23  begin
24
25    Stages:
26      for i in 7 downto 0 generate
27
28        LowBit:
29          if i = 0 generate
30            FA: FullAdder port map
31                  (A(0), B(0), Cin, C(0), Sum(0));
32          end generate;
33
34        OtherBits:
35          if i /= 0 generate
36            FA: FullAdder port map
37                  (A(i), B(i), C(i-1), C(i), Sum(i));
38          end generate;
39
40      end generate;
41
42    Cout <= C(7);
43
44  end;
45
46  ------------------------------------------------
```

## ❑ Implementation

This structural description uses a new facility, the *generate statement*. Its value was motivated by one of the exercises in the last section. Without the generate statement, you had to create one component instance per bit of the latch; thus the size of the latch description (measured in terms of the number of statements) is linear with the size of the latch (measured in terms of the number of bits).

In contrast, use of the generate statement makes the size of the adder description independent of the size of the adder. To create a 32-bit adder, a textual substitution of "31" for all occurrences of "7" is all that is required.

There are two kinds of generate statement. One, the *for generate* (lines 25 through 40), is used to replicate statements in a concurrent environment. (Contrast the for generate with the for loop—the for generate causes the repeated elaboration of concurrent statements; the for loop causes the repeated execution of sequential statements).

The for generate begins with a label (line 25—an identifier followed by a colon). The label is followed by the reserved word **for**, an iterator, a range, and the reserved word **generate** (line 26). Then, a set of concurrent statements to be replicated is given (lines 28 through 38). Finally, the reserved words **end** and **generate**, followed by a semicolon terminates the generate statement. Occurrences of the iterator ("i" in this example) within the concurrent statements are replaced with the current value of the iterator within each "iteration" of the generate statement.

The other kind of generate statement is the *if generate*. (The adder uses two if generates; one on lines 28 through 32 and the other on lines 34 through 38.) The if generate causes the conditional elaboration of concurrent statements. (Again, contrast the if generate with the if statement—which we have not yet seen. The if statement conditionally executes sequential statements.)

The if generate also begins with a label (line 28 and line 34), but is then followed by the reserved word **if**. Then a conditional expression that controls embedded statement generation and the reserved word **generate** follows (line 29 and line 35). The remainder of the if generate is identical to the body and end of the for generate: a set of concurrent statements to be conditionally generated (lines 30 and 31 and lines 36 and 37), the reserved words **end** and **generate**, and a semicolon.

Notice how the ripple carry is modeled. An eight-bit signal, C is declared on line 21. Each bit, *n*, serves as the carry between stages *n* and *n+1* (line 37). The carry input of the zeroth stage received the carry input, "Cin", of the full adder (line 31). Finally, the adder's carry output, "Cout" is driven from the most-significant bit of "C" on line 42.

The generate statements in this example are used to create a one-dimensional structure. By nesting for generates, arbitrary n-dimensional structures can be created. By utilizing if generates appropriately, "sparse" structures (such as triangular arrays) of arbitrary dimension can also be created.

```
47  ------------------------------------------------
48
49  entity Test_Adder8 is end;
50
51  ------------------------------------------------
52
53  architecture Driver of Test_Adder8 is
54
55    component Adder8
56      port (A, B: in      Bit_Vector(7 downto 0);
57            Cin:  in      Bit;
58            Cout: out     Bit;
59            Sum:  out     Bit_Vector(7 downto 0));
60    end component;
61
62    signal A, B, Sum: Bit_Vector(7 downto 0);
63    signal Cin, Cout: Bit := '0';
64
65  begin
66
67    UUT: Adder8 port map (A, B, Cin, Cout, Sum);
68
69    Stimulus:
70      process
71          variable Temp: Bit_Vector(7 downto 0);
72      begin
73          Temp := "00000000";
74          for i in 1 to 32 loop
75            if i mod 2 /= 1 then
76              A <= Temp;
77              B <= "00000001";
78            else
79              B <= Temp;
80              A <= "00000001";
81            end if;
82            wait for 1 ns;
83            Temp := Sum;
84          end loop;
85          wait; -- to terminate simulation
86      end process;
87
88  end;
89
90  ------------------------------------------------
```

## ❑ Verification

This test bench contains no response checking. It applies the tests $n + 1$ and $1 + n$ for $1 \leq n \leq 31$.

The one new VHDL construct is the *if statement* (lines 75 through 81). The if statement allows the conditional execution of sequential statements. The form of the if statement is

```
if condition-1 then
    sequential-statements-1
{ elsif condition-2 then
    sequential-statements-2 }
[ else
    sequential-statements-3 ]
end if;
```

(The braces, "{ … }", indicate that the enclosed text can be replicated zero or more times. The brackets, "[ … ]", indicate that the enclosed text is optional.)

When the if statement is executed, the conditions are checked, in textual order, until one that is true is found. The corresponding sequential statements are then executed. If none of the conditions are true and if the else clause (that begins with the reserved word **else**) is present, then the sequential statements within the else clause are executed.

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
  17 - 24  : /test_adder8(driver).sum

      Time(fs) + Cycle    17  18  19  20  21  22  23  24
----------------------   --- --- --- --- --- --- --- ---
                  0+ 0:  '0' '0' '0' '0' '0' '0' '0' '0'
                  0+ 1: *'0'*'0'*'0'*'0'*'0'*'0'*'0'*'0'
                  0+ 2:  '0' '0' '0' '0' '0' '0' '0'*'1'
...
            1000000+ 2:  '0' '0' '0' '0' '0' '0' '0'*'0'
            1000000+ 3:  '0' '0' '0' '0' '0' '0'*'1' '0'
...
            2000000+ 2:  '0' '0' '0' '0' '0' '0'*'0'*'1'
            2000000+ 3:  '0' '0' '0' '0' '0'*'1'*'1' '1'
            2000000+ 4:  '0' '0' '0' '0' '0'*'0' '1' '1'
...
            3000000+ 2:  '0' '0' '0' '0' '0' '0'*'1'*'0'
            3000000+ 3:  '0' '0' '0' '0' '0' '0'*'0' '0'
            3000000+ 4:  '0' '0' '0' '0' '0'*'1' '0' '0'
...
            4000000+ 2:  '0' '0' '0' '0' '0'*'0'*'1'*'1'
            4000000+ 3:  '0' '0' '0' '0'*'1'*'1'*'0' '1'
            4000000+ 4:  '0' '0' '0' '0'*'0' '1' '0' '1'
...
            5000000+ 2:  '0' '0' '0' '0' '0'*'1' '0'*'0'
            5000000+ 3:  '0' '0' '0' '0' '0' '1'*'1' '0'
...
            6000000+ 2:  '0' '0' '0' '0' '0'*'1'*'0'*'1'
            6000000+ 3:  '0' '0' '0' '0' '0'*'0'*'1' '1'
            6000000+ 4:  '0' '0' '0' '0'*'1'*'1' '1' '1'
            6000000+ 5:  '0' '0' '0' '0'*'0' '1' '1' '1'
...
            7000000+ 2:  '0' '0' '0' '0' '0'*'1'*'1'*'0'
            7000000+ 3:  '0' '0' '0' '0' '0' '1'*'0' '0'
            7000000+ 4:  '0' '0' '0' '0' '0'*'0' '0' '0'
            7000000+ 5:  '0' '0' '0' '0'*'1' '0' '0' '0'
...
            8000000+ 2:  '0' '0' '0' '0'*'0'*'1'*'1'*'1'
            8000000+ 3:  '0' '0' '0'*'1'*'1'*'0'*'0' '1'
            8000000+ 4:  '0' '0' '0'*'0' '1' '0' '0' '1'
...
            9000000+ 2:  '0' '0' '0' '0'*'1' '0' '0'*'0'
            9000000+ 3:  '0' '0' '0' '0' '1' '0'*'1' '0'
...
           10000000+ 2:  '0' '0' '0' '0'*'1' '0'*'0'*'1'
           10000000+ 3:  '0' '0' '0' '0' '1'*'1'*'1' '1'
           10000000+ 4:  '0' '0' '0' '0' '1'*'0' '1' '1'
...
           11000000+ 2:  '0' '0' '0' '0'*'1' '0'*'1'*'0'
           11000000+ 3:  '0' '0' '0' '0' '1' '0'*'0' '0'
           11000000+ 4:  '0' '0' '0' '0' '1'*'1' '0' '0'
...
           12000000+ 2:  '0' '0' '0' '0'*'1'*'0'*'1'*'1'
           12000000+ 3:  '0' '0' '0' '0'*'0'*'1'*'0' '1'
           12000000+ 4:  '0' '0' '0'*'1'*'1' '1' '0' '1'
           12000000+ 5:  '0' '0' '0'*'0' '1' '1' '0' '1'
...
           13000000+ 2:  '0' '0' '0' '0'*'1'*'1' '0'*'0'
           13000000+ 3:  '0' '0' '0' '0' '1' '1'*'1' '0'
...
```

## ❑ Running the Example

Since the test bench for this example is essentially a counter, using the eight-bit adder to increment the count, it will be easy to examine the output waveforms to verify that the adder is functioning correctly—we need only look at the "sum" output signal. Once again, we will edit the format control file to inhibit formatting for all signals except the eight bits of "sum". We can also use the "-nonames" switch to cause *format* to inhibit printing of the signal names in the formatted output, in order to further minimize the width of the file.

The commands to run this example are as follows:

```
% cp $CLSI/examples/tutorial/adder8.vhdl .
% compile adder8
% build test_adder8
% simulate test_adder8 -trace
% format test_adder8 -preview
% #edit test_adder8.control to display only signals 17 - 24
% format test_adder8 -nonames
% more test_adder8.wave
```

At left is a copy of the `wave` file you have just produced—truncated to fit on a single page. Verify the correct operation of the adder.

## ❑ Additional Exercises

1. Build a 32-bit adder.
2. Rewrite the test bench to test for all combinations of $0 \le A \le 31$ and $0 \le B \le 31$.
3. Using if generates, eliminate the need for C(7) by driving Cout with the carry output of the most-significant stage of the adder.
4. Build a lookahead-carry, eight-bit adder using random logic and the dataflow style of description.

## Modeling Behavior—A Shift Register

| CLK | CLR | LD | SH | DIR | Q |
|---|---|---|---|---|---|
| X | 1 | X | X | X | 0 |
| 0 | 0 | X | X | X | Q* |
| 1 | 0 | X | X | X | Q* |
| ↑ | 0 | 0 | 0 | X | Q* |
| ↑ | 0 | 1 | X | X | D |
| ↑ | 0 | 0 | 1 | 0 | 0 & Q*(1 to *n*-1) |
| ↑ | 0 | 0 | 1 | 1 | Q*(2 to *n*) & 0 |

## ❑ Specification

Our next model is a shift register of configurable width. That is, it is a shift register whose D input is $m$ bits wide and whose Q output is $n$ bits wide, subject to the restriction that $m \leq n$.

We see by examination of the truth table opposite that the shift register has an asynchronous, active-high clear that overrides all other inputs. Moreover, when the clock is stable (either low or high), no activity occurs—the notation Q* indicates that the previous state of the shift register is asserted on the Q output. Only on the rising edge of the clock can a load or shift occur. Then, if both LD and SH are low, no activity occurs. However, if LD is high, then irrespective of any other inputs the current value of D is latched into the register (with the most-significant $n$-$m$ bits being set to '0'). If LD is instead low and SH is high, then the current value of the shift register is shifted one bit either to the right (if DIR is low) or to the left (if DIR is high). The vacated bit position in any case is filled with '0'.

```
 1  -------------------------------------------------
 2
 3  entity ShiftN is
 4    port (CLK:    in      Bit;
 5          CLR:    in      Bit;
 6          LD:     in      Bit;
 7          SH:     in      Bit;
 8          DIR:    in      Bit;
 9          D:      in      Bit_Vector;
10          Q:      out     Bit_Vector);
11
12  begin
13
14    assert (D'Length <= Q'Length)
15      report "Input D must not be wider than output Q"
16        severity Failure;
17
18  end ShiftN;
19
20  -------------------------------------------------
21
22  architecture Behavior of ShiftN is
23  begin
24
25   Shifter:
26
27    process (CLR, CLK)
28
29      subtype InBits  is Natural range D'Length-1 downto 0;
30      subtype OutBits is Natural range Q'Length-1 downto 0;
31      variable State: Bit_Vector(OutBits);
32
33    begin
34
35      if CLR = '1' then
36
37        State := (others => '0');
38        Q <= State after 3ns;
39
40      elsif CLK'Event and CLK='1' then
41
42        if LD = '1' then
43
44          State := (others => '0');
45          State(InBits) := D;
46          Q <= State after 5ns;
47
48        elsif SH = '1' then
49
50          case DIR is
51            when '0'  =>
52              State := '0' & State(State'Left downto 1);
53            when '1'  =>
54              State := State(State'Left-1 downto 0) & '0';
55          end case;
56          Q <= State after 7ns;
57
58        end if;
59
60      end if;
61
62    end process;
63
64  end;
65
66  -------------------------------------------------
```

## ❑ Implementation

Here is the VHDL model for our shift register, a *behavioral description*, so-called because the model is described as an algorithm, not as a physical or logical structure.

Recall that we wished to be able to configure the number of bits in the shift register. VHDL allows this capability through the use of *unconstrained ports*. Examine the declaration of the interface signals "D" and "Q" on lines 9 and 10 on the page opposite. Notice that they are of type Bit_Vector, but that the number of elements (called the *constraint*) is not specified. (Contrast these declarations with the declarations of "D" and "Q" for the Latch, where a constraint is supplied.) The lack of constraint has two implications:

1. The model can be written to handle any number of bits. (We will see how to do so shortly.)

2. The constraint must be supplied when this entity is bound to a component instance, as we will see in a later example.

Another new feature is the *entity statement part* (lines 12 through 16), which allows characteristics of the entity (such as timing checks or, as we have here, relationships between the characteristics of ports) to be checked with either assertion statements or *passive processes* (processes not containing any signal assignment statements).

We use an assert statement within the entity statement part to ensure that the width of the D input is not greater than the width of the Q output. This assert has several new features. The first is the use of the *predefined attribute* "Length", which supplies the number of elements in an array. So, "D" being an array (a Bit_Vector), "D'Length" (read "dee tick length") returns the number of elements (Bits) in the array used as the *prefix* (in this case, "D") to the attribute. (The "Length" attribute can be used only on arrays.) Since this name is not evaluated until after the ShiftN entity has been bound to a component instance, the number of bits is known.

The other new feature of the assert is its *severity clause* (line 16). Should the assertion expression be false when it is evaluated (referred to as an *assertion violation*), not only will the report string be delivered, but the severity is also displayed. The severity expression (following the reserved word **severity**) can be one of the following values: Note, Warning, Error, or Failure—the default (in the absence of a severity clause) is Error. When an assertion violation occurs, in addition to displaying the report string and severity expression, if the severity is Error or Failure the VSS causes a breakpoint to occur, returning control to the modeler. Simulation cannot be continued at this point, but other commands to examine values, etc., can be issued. In addition, when the VSS is terminated, VSS returns a "failure" indication to the host operating system.

The architecture (lines 22 through 62) also has much that is new. Firstly, we are using a process to describe the behavior of a model. We had previously used processes only within the test bench, but as processes can describe arbitrary behavior, we can also use them to describe the model itself.

A new feature of the process statement itself is its *sensitivity list* (line 27). The sensitivity list is a parenthesized list of signals to which the process is *sensitive*.

A process is sensitive to a signal when a change on the signal causes the process to execute. So, this process will execute whenever the signals "CLR" or "CLK" change their value. The process executes by executing its statements (lines 35 through 60) in order, until the last statement in the process has finished executing. The process then awaits another change on a signal to which it is sensitive.

Within the process are two *subtype declarations* (lines 29 and 30). Subtypes are additional constraints on an existing type or subtype. For example, in the VSS, the predefined type Integer is declared as

```
type Integer is range -2147483648 to 2147483647;
```

and the predefined subtype Natural is declared as

```
subtype Natural is Integer range 0 to 2147483647;
```

which means that Natural is an Integer that must be non-negative. Hence, the subtype "InBits", declared on line 29, is an Integer, descending range that runs from the most-significant bit number of the input "D" down to 0. Similarly, the subtype "OutBits", declared on line 30, is an Integer, descending range that runs from the most-significant bit number of the output "Q" down to 0.

One of these subtypes is used immediately. The internal state of the shift register is held in the variable "State", declared on line 31. Notice its type; it is Bit_Vector, but the numbering of the bits (and hence the number of bits) is given as the subtype OutBits. This means that "State" has the same number of bits as "Q."

Line 37 contains the signal assignment used to clear State. We cannot use a string containing all zeros, for we do not know how many bits are in State, and VHDL requires that the number of bits on each side of the assignment match. Fortunately, VHDL gives us a way of constructing Bit Vectors (and other arrays) of whatever length is necessary to match. Using an *aggregate* containing an *others association*, we can assign to State a vector of however many zeros are in State. (An aggregate is a parenthesized list of values; here, we have a single value: "others => '0'". The notation "others =>" indicates that any previously unmentioned elements in the aggregate—here, all of them—are to be filled with the value after the arrow.)

Line 40 demonstrates the use of the elsif clause of the if statement, discussed previously. It also introduces another predefined attribute, "Event". This attribute, whose prefix must be a signal, returns "True" at those times when the prefix signal has just changed value and "False" otherwise. Hence, the expression "CLK'Event and CLK='1'" means "CLK has just changed *and* CLK is now high." Therefore, the expression is true precisely on the rising edges of "CLK."

Lines 42 through 58 demonstrate the use of an if statement nested within another if statement. Because of the use of the bracketing reserved words **elsif** and **end if**, there is no ambiguity over the association of nested ifs, as there are in some languages.

Line 45 shows how a slice name can be used to assigned to only as many bits of State as there are in D. Because of the use of descending ranges with identical right bounds in defining the subtypes on lines 29 and 30, the least-significant bits of State are assigned.

Lines 50 through 55 introduce VHDL's *case statement*. The case statement operates by evaluating the expression appearing between the reserved words **case** and **is**. Then, the expression's value is located among the case statement's *when clauses* (lines 51 and 53)—there must be exactly one that matches the expression's value. The statements following the "=>" are then executed. Once the final statement in the appropriate when clause finishes executing, the case statement terminates; it does not "fall through" to the next clause the way that C's switch statement or Fortran's computed goto do.

Finally, both lines 52 and 54 demonstrate the use of another predefined attribute, "Left." Left determines the left-most index value of the prefix array. So, "State'Left" is the same as "Q'Length-1."

```
 67  -------------------------------------------------
 68
 69  entity Test_ShiftN is end;
 70
 71  -------------------------------------------------
 72
 73  architecture Driver of Test_ShiftN is
 74
 75    component ShiftN
 76      port (CLK:  in      Bit;
 77            CLR:  in      Bit;
 78            LD:   in      Bit;
 79            SH:   in      Bit;
 80            DIR:  in      Bit;
 81            D:    in      Bit_Vector;
 82            Q:    out     Bit_Vector);
 83    end component;
 84
 85    signal CLK, CLR, LD, SH, DIR: Bit;
 86    signal D: Bit_Vector(1 to 4);
 87    signal Q: Bit_Vector(8 downto 1);
 88
 89  begin
 90
 91    UUT: ShiftN port map (CLK, CLR, LD, SH, DIR, D, Q);
 92
 93    Stimulus:
 94       process
 95       begin
 96
 97          -- clear the register
 98
 99          CLR <= '1', '0' after 10ns;
100          wait for 10ns;
101
102          -- load the register
103
104          D <= "1110";
105          LD <= '1', '0' after 10ns;
106          CLK <= '0', '1' after 3 ns;
107          wait for 10 ns;
108
109          -- left shift the pattern
110
111          SH <= '1';
112          DIR <= '1';
113          for i in 1 to 5 loop
114            CLK <= '0', '1' after 3ns;
115            wait for 10ns;
116          end loop;
117
118          -- right shift the pattern
119
120          DIR <= '0';
121          for i in 1 to 8 loop
122            CLK <= '0', '1' after 3ns;
123            wait for 10ns;
124          end loop;
125
126          wait;
127       end process;
128
129  end;
130
131  -------------------------------------------------
```

## ❑ Verification

The shift register test bench on the page opposite instantiates a unit under test having a four-bit D input and an eight-bit Q output. Note that the component declaration (lines 75 through 82) does not constrain the D and Q ports (lines 81 and 82); the constraints are applied when the component is instantiated.

The constraints are applied by inference. Note the declaration of the internal signal D (on line 86); it is four-bits wide with an ascending range. Similarly, Q is declared (on line 87) as an eight-bit wide signal with a descending range of 8 downto 1. When these signals are associated with the shifter component's ports on line 91, this constraint information is passed to the shift register model when the entity and architecture are elaborated.

Recall that in the model, we assumed (in the declaration of the subtypes "InBits" and "OutBits") that we would be indexing D and Q in descending fashion from length-1 downto 0. However, in the test bench, Q is declared as 8 downto 1, and, worse, D is declared from 1 to 4. This is not a problem. VHDL uses the *matching element rule* to arbitrate such situations. As long as there are the same number of elements in the ranges, the leftmost element of one range is matched with the leftmost element of the other, the next-leftmost with the next-leftmost, and so forth. Thus, "D(1)" in the test bench is "D(3)" in the model, "D(2)" in the test bench is "D(2)" in the model, "D(3)" in the test bench is "D(1)" in the model, and "D(4)" in the test bench is "D(0)" in the model. Similarly, "Q(8)" in the test bench is "Q(7)" in the model, "Q(7)" in the test bench is "Q(6)" in the model, etc.

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
  10 - 17  : /test_shiftn(driver).q

                          10   11   12   13   14   15   16   17
      Time(fs) + Cycle  q(8) q(7) q(6) q(5) q(4) q(3) q(2) q(1)
----------------------  ---- ---- ---- ---- ---- ---- ---- ----
                   0+ 0:  '0'  '0'  '0'  '0'  '0'  '0'  '0'  '0'
...
             3000000+ 0: *'0' *'0' *'0' *'0' *'0' *'0' *'0' *'0'
...
            18000000+ 0: *'0' *'0' *'0' *'0' *'1' *'1' *'1' *'0'
...
            30000000+ 0: *'0' *'0' *'0' *'1' *'1' *'1' *'0' *'0'
...
            40000000+ 0: *'0' *'0' *'1' *'1' *'1' *'0' *'0' *'0'
...
            50000000+ 0: *'0' *'1' *'1' *'1' *'0' *'0' *'0' *'0'
...
            60000000+ 0: *'1' *'1' *'1' *'0' *'0' *'0' *'0' *'0'
...
            70000000+ 0: *'1' *'1' *'0' *'0' *'0' *'0' *'0' *'0'
...
            80000000+ 0: *'0' *'1' *'1' *'0' *'0' *'0' *'0' *'0'
...
            90000000+ 0: *'0' *'0' *'1' *'1' *'0' *'0' *'0' *'0'
...
           100000000+ 0: *'0' *'0' *'0' *'1' *'1' *'0' *'0' *'0'
...
           110000000+ 0: *'0' *'0' *'0' *'0' *'1' *'1' *'0' *'0'
...
           120000000+ 0: *'0' *'0' *'0' *'0' *'0' *'1' *'1' *'0'
...
           130000000+ 0: *'0' *'0' *'0' *'0' *'0' *'0' *'1' *'1'
...
```

## ❑ Running the Example

Previous models have exhibited strict stimulus–response behavior, where the examination of signal values over time provides all necessary information about the execution of the model. Behavioral description not only raises the modeling style above this level, but also introduces the need for additional debugging facilities. *Breakpoints*, much like those in software debuggers, are included in the VSS for the purpose of monitoring the execution of processes. Another feature of VSS is the ability to interactively examine the values of signals and variables as they change during simulation.

To use breakpoints, you must use run the simulation in interactive mode. Enter the following commands to build the model, get the simulation started, and set an initial breakpoint:

```
% cp $CLSI/examples/tutorial/shiftn.vhdl .
% compile shiftn -debug
% build test_shiftn
% simulate test_shiftn -interactive -trace
Command> initialize
Command> show processes
Command> break process /test_shiftn/uut/shifter
Command> show breakpoints
```

Note that a breakpoint was implicitly inserted as a result of the *initialize* command in order to stop simulation at the end of initialization. A second breakpoint was inserted as a result of the *break process* command. Each time process "Shifter" resumes, control will return to the user and a "Command>" prompt will be displayed on the screen.

At this point you can single-step through the shift register's process using the *step* command. You can enter the *help* command to learn about various kinds of *step* commands that are available in interactive mode. Execute the following sequence of commands:

```
Command> go
Command> step   # do this several times
```

Note that the *step* command takes you through the process execution one statement at a time. Note also that, in some points, when you *step* through a VHDL *wait* statement, you step into the execution of another process—in this case, the assertion statement in the entity interface.

To aid in debugging the execution of a VHDL description, VSS also allows you to examine the values of signals, variables, and constants as you step through the execution of a description. Use the *help* command to learn how to use the *examine* command. You can also refer to the VSS Command Reference for a somewhat more detailed description of the commands available.

Start out with the following sequence of commands:

```
Command> cancel break *
Command> break line 50
Command> go
```

You are now about to execute the assignment to variable State on line 52 or the assignment to variable State on line 54 of the source file "shiftn.vhdl". Execute the following commands to see the effect of the assignment:

```
Command> step
Command> examine State
Command> step
Command> examine State
```

Now you are about to execute the assignment to signal Q on line 56. Execute the following commands to see the effect of the assignment:

```
Command> examine Q
Command> step
Command> examine Q
```

Note that the signal did not change value. Recall that assignments to signals always involve at least a minimal delay; the assignment will not take effect until the process suspends and time advances.

To complete the simulation in non-interactive mode, you must first remove all breakpoints:

```
Command> cancel break *
Command> go
```

At left is a formatted waveform file showing the values of output signal "q". Finish up this simulation by using the *format* command to produce a similar file from the results of your simulation.

## ❑ Additional Exercises

1. Instantiate a 32-bit wide shift register in the test bench. How must the code be changed?

2. Instantiate a shift register that violates the requirement that D'Length <= Q'Length. At what point is the problem detected?

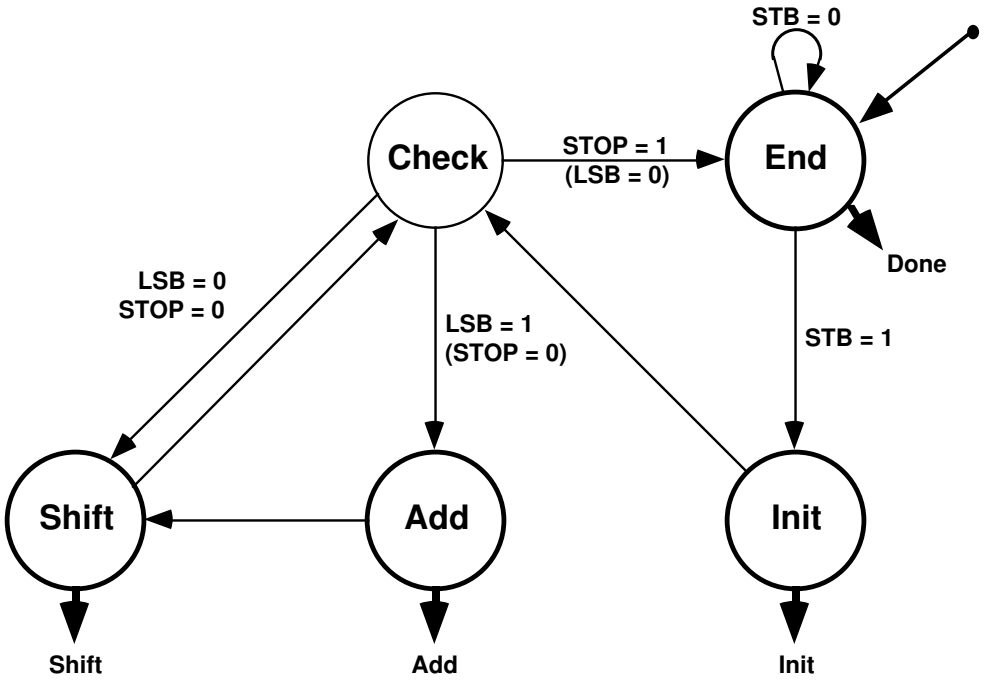
3. Modify the shift register model and test bench to incorporate a FILL input to the shift register. The FILL bit is shifted into the vacated position during a shift operation.

4. Modify the shift register to incorporate two shift function bits, SF1 and SF2, that control the shift register as follows:

| SF1 | SF2 | Action |
|---|---|---|
| '0' | '0' | Shift a '0' into the vacated position (the current behavior). |
| '0' | '1' | Shift a '1' into the vacated position. |
| '1' | '0' | Rotate: shift the bit about to be discarded into the vacated position. |
| '1' | '1' | Inverse rotate: shift the inverse of the discarded bit into the vacated position. |

## Modeling a Finite-State Machine—A Controller

| State | Action | Next State |
|---|---|---|
| End: | if STB | -> Init: |
| Init: | load SRA, SRB<br>clear ACC | -> Check: |
| Check: | if Stop<br>if not Stop and LSB = 0<br>if LSB = 1 | -> End:<br>-> Shift:<br>-> Add: |
| Add: | load ACC | -> Shift: |
| Shift: | shift SRA, SRB | -> Check: |



## ❑ Specification

Our next model is of a finite-state machine controller for the shift-and-add multiplier. The multiplier accepts two inputs, A and B, and contains an accumulator that is initialized to zero. Then, while not all of the bits of A are zero, it repeatedly performs the following steps:

1. If the least-significant bit of A is non-zero, add B to the accumulator.
2. Set A := A div 2 and B := B * 2 by shifting A one bit to the right and B one bit to the left.

When A = 0, the accumulator contains A * B.

We need a controller for such an adder. We assume that it has three data inputs:

STB: strobes A and B and initializes the multiplier,

LSB: the least-significant bit of A, and

STOP: all of the bits of A are zero,

a clock input, and four outputs:

INIT: initialize the accumulator,

SHIFT: shift A one bit to the right and B one bit to the left,

ADD: add the current B to the accumulator, and

DONE: indicate operation is complete.

The state table and state transition diagram for this controller are shown on the page opposite. The FSM is initially in the End state, in which the output signal DONE is asserted. When the STB signal is raised, we enter the Init state, which raises the INIT signal to load the A and B registers and clear the accumulator. We then proceed to the Check state.

If we're in the Check state, and if the STOP signal is high (which implies that LSB = '0'), we go back to the Init state. Otherwise, if LSB = '0' we go to the Shift state, else if LSB = '1', we go to the Add state. No output signals are asserted in this state.

If we're in the Add state, we assert the ADD signal, which causes the current contents of the B register to be added to the accumulator. We then proceed to the Shift state.

If we're in the Shift state, we assert the SHIFT signal, which causes the A register to shift one bit to the right and the B register to shift one bit to the left. We then proceed to the Check state.

```
 1  -------------------------------------------------
 2
 3  entity Controller is
 4    port (STB, CLK, LSB, Stop:    in      Bit;
 5          Init, Shift, Add, Done: out     Bit);
 6  end Controller;
 7
 8  -------------------------------------------------
 9
10  architecture FSM of Controller is
11
12    type States is (InitS, CheckS, AddS, ShiftS, EndS);
13    signal State: States := EndS;
14
15  begin
16
17    -- Drive control outputs based upon State
18
19    Init  <= '1' when State = InitS  else '0';
20    Add   <= '1' when State = AddS   else '0';
21    Shift <= '1' when State = ShiftS else '0';
22    Done  <= '1' when State = EndS   else '0';
23
24    -- Determine Next State from control inputs
25
26    StateMachine:
27
28      process (CLK)
29      begin
30        if CLK'Event and CLK = '0' then
31          case State is
32            when InitS =>
33              State <= CheckS;
34            when CheckS =>
35              if LSB = '1' then
36                State <= AddS;
37              elsif Stop = '0' then
38                State <= ShiftS;
39              else
40                State <= EndS;
41              end if;
42            when AddS =>
43              State <= ShiftS;
44            when ShiftS =>
45              State <= CheckS;
46            when EndS =>
47              if STB = '1' then
48                State <= InitS;
49              end if;
50          end case;
51        end if;
52      end process;
53
54  end;
55
56  -------------------------------------------------
```

## ❑ Implementation

Here is a *finite-state machine description* of the controller. The entity interface is straightforward. The architecture is written as a single-process behavior, but several features particularize this behavioral description as a finite-state machine-style of description.

Firstly, we use an *enumeration type* to represent the possible states of the machine. An enumeration type is a type whose possible values are described as identifiers or character literals. For example, the predefined type "Bit" is described as an enumeration type. The *enumeration type declaration* on line 12 declares these states, InitS, CheckS, AddS, ShiftS, and EndS. The signal State, declared on line 13, is the state register; it is initialized to the EndS state.

Lines 19 through 22 use a new kind of concurrent signal assignment statement, the *conditional signal assignment statement*, to drive the four output signals depending on the FSM's current state. For example, look at line 19. Whenever State is InitS, the Init signal is '1'; otherwise, it is '0'. This signal assignment statement (as well as the ones on lines 20 to 22) executes whenever the State register takes on a new state.

The state transition is implemented by the process on lines 28 through 52, which executes whenever CLK has an edge. The if statement beginning on line 30 filters out the rising edges of CLK, so the body of the if statement (which constitutes the rest of the process) executes only on the falling edge of CLK.

On the falling edge of CLK, the case statement beginning on line 31 examines the current state and branches to the appropriate list statements. There, the remainder of the FSM's inputs are examined as needed to determine the appropriate next state.

```
 56  -------------------------------------------------
 57
 58  entity Test_Controller is end;
 59
 60  -------------------------------------------------
 61
 62  architecture Driver of Test_Controller is
 63
 64    component Controller
 65      port (STB, CLK, LSB, Stop:  in      Bit;
 66            Init, Shift, Add, Done:       out     Bit);
 67    end component;
 68
 69    signal STB, LSB, Done, Init, Shift, Add, Stop: Bit;
 70    signal CLK, ENA: Bit := '1';
 71
 72  begin
 73
 74    UUT: Controller port map
 75           (STB, CLK, LSB, Stop, Init, Shift, Add, Done);
 76
 77    Clock:  CLK <= not (CLK and ENA) after 10 ns;
 78    Enable: ENA <= '0' after 500 ns;
 79
 80    Stimulus:
 81      process
 82        variable Val: Bit_Vector(7 downto 0);
 83      begin
 84        wait until CLK'Event and CLK='1';
 85        STB <= '1', '0' after 20 ns;
 86        Val := "01110101";
 87        loop
 88          LSB <= Val(0);
 89          Stop <= not (Val(7) or Val(6) or Val(5) or Val(4) or
 90                       Val(3) or Val(2) or Val(1) or Val(0));
 91          wait until CLK'Event and CLK='1';
 92          exit when Done = '1';
 93          if Shift = '1' then
 94            Val := '0' & Val(7 downto 1);
 95          end if;
 96        end loop;
 97        wait;
 98      end process;
 99
100  end;
101
102  -------------------------------------------------
```

## ❑ Verification

Here is the test bench for the FSM controller. Lines 77 and 78 implement a gated clock that runs from 0 to 500 ns. During this interval, it is a 20 ns clock with a 50% duty cycle.

The stimulus generator, on lines 80 through 98, emulates the data path of the multiplier, so that it can exercise the controller. It uses two wait statements (on lines 84 and 91) to synchronize with the clock; these wait statements incorporate a new feature, the *until clause*, consisting of the reserved word **until** followed by a Boolean expression.

The until clause affects the behavior of the wait statement as follows. The process containing the wait statement is sensitized to all signals contained in the until clause's expression (in this case, CLK) while the wait statement is active. Whenever there is a change on one of these signals, the expression is evaluated. If it is true, the wait statement completes, and the enclosing process continues its execution. If the expression is false, the wait statement continues to wait until there is another event on one of the signals to which it is sensitive. Thus, these wait statements wait for a rising edge on CLK.

The second wait statement is enclosed in a new form of loop statement, the *indefinite loop* (lines 87 through 96). This loop has no iteration control; thus, in the absence of other control mechanisms, which we will shortly see, it never terminates.

One such "other control mechanism" is found on line 92, the *exit statement*. This exit statement has a Boolean expression (Done = '1') following the reserved word **when**. Whenever the exit statement is executed, the expression is evaluated. If it is true, the innermost enclosing loop is exited (thus, we would continue execution on line 97 in this case). If the expression is instead false, the execution of the loop proceeds as normal. (The *when clause* of the exit statement is optional; thus the statement `exit;` causes an unconditional exit from the innermost enclosing loop.)

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
       4   : /test_controller(driver).init
       5   : /test_controller(driver).shift
       6   : /test_controller(driver).add
       9   : /test_controller(driver).uut/controller(fsm).state

                            4     5    6       9
      Time(fs) + Cycle   init shift  add   state
----------------------   ---- ----- ---- -------
                  0+ 0:  '0'   '0'  '0'    ends
                  0+ 1: *'0' * '0' *'0'    ends
...
           30000000+ 1:  '0'   '0'  '0' * inits
           30000000+ 2: *'1' * '0' *'0'   inits
...
           50000000+ 1:  '1'   '0'  '0' *checks
           50000000+ 2: *'0' * '0' *'0'  checks
...
           70000000+ 1:  '0'   '0'  '0' *  adds
           70000000+ 2: *'0' * '0' *'1'    adds
...
           90000000+ 1:  '0'   '0'  '1' *shifts
           90000000+ 2: *'0' * '1' *'0'  shifts
...
          110000000+ 1:  '0'   '1'  '0' *checks
          110000000+ 2: *'0' * '0' *'0'  checks
...
          130000000+ 1:  '0'   '0'  '0' *shifts
          130000000+ 2: *'0' * '1' *'0'  shifts
...
          150000000+ 1:  '0'   '1'  '0' *checks
          150000000+ 2: *'0' * '0' *'0'  checks
...
          170000000+ 1:  '0'   '0'  '0' *  adds
          170000000+ 2: *'0' * '0' *'1'    adds
...
          190000000+ 1:  '0'   '0'  '1' *shifts
          190000000+ 2: *'0' * '1' *'0'  shifts
...
          210000000+ 1:  '0'   '1'  '0' *checks
          210000000+ 2: *'0' * '0' *'0'  checks
...
          230000000+ 1:  '0'   '0'  '0' *shifts
          230000000+ 2: *'0' * '1' *'0'  shifts
...
          250000000+ 1:  '0'   '1'  '0' *checks
          250000000+ 2: *'0' * '0' *'0'  checks
...
          270000000+ 1:  '0'   '0'  '0' *  adds
          270000000+ 2: *'0' * '0' *'1'    adds
...
          290000000+ 1:  '0'   '0'  '1' *shifts
          290000000+ 2: *'0' * '1' *'0'  shifts
...
          310000000+ 1:  '0'   '1'  '0' *checks
          310000000+ 2: *'0' * '0' *'0'  checks
...
          330000000+ 1:  '0'   '0'  '0' *  adds
          330000000+ 2: *'0' * '0' *'1'    adds
...
```

## ❑ Running the Example

To reproduce the simulation results on the page opposite, simulate the controller and display signal numbers 4, 5, 6, and 9. This time, simulate in interactive mode, and use the *show* commands to obtain selected information about the simulation. Try the following sequence of instructions:

```
% cp $CLSI/examples/tutorial/controller.vhdl .
% compile controller -debug
% build test_controller
% simulate test_controller -interactive -trace
Command> show hierarchy
Command> show processes
Command> elaborate
Command> show hierarchy
Command> show processes
Command> show signals
Command> initialize
Command> show breakpoints
Command> break process /test_controller/uut/StateMachine
Command> show breakpoints
Command> step process
Command> show time
Command> step cycle
Command> step process
Command> show time
Command> step cycle
Command> step process
Command> show time
Command> cancel break *
Command> show breakpoints
Command> go
Command> quit
```

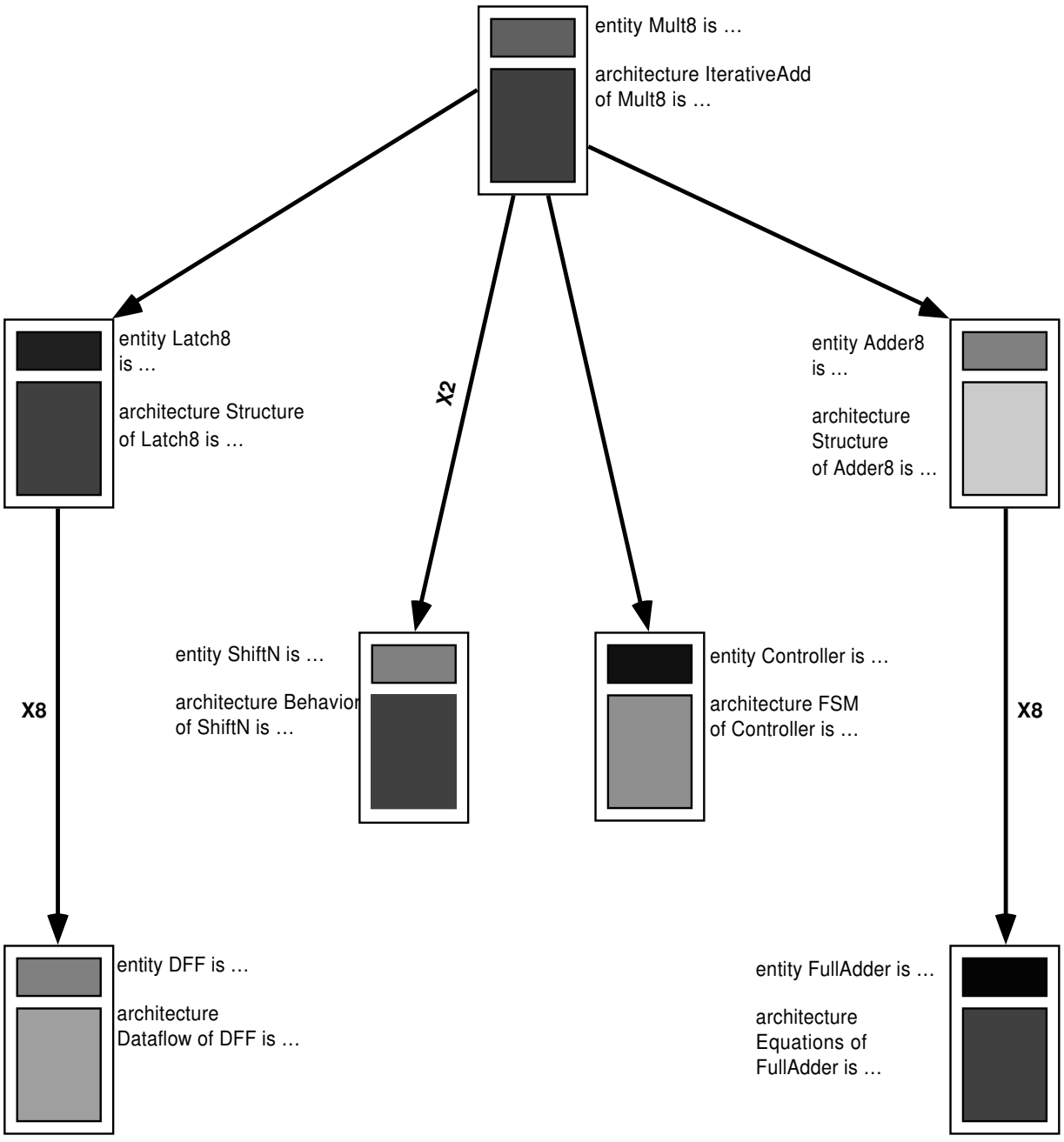
When you are finished with the simulation, use the *format* command as before to produce the formatted waveform file. If you want to make the result look like that displayed on the opposite page, remember to edit the control file first. Note that the results opposite have been truncated to fit the page.

## ❑ Additional Exercises

1. Rewrite the process in the controller model's architecture to eliminate the use of the sensitivity list; instead, use a wait statement with an until clause. Resimulate, using a breakpoint at the beginning of the process. Do you notice any difference in the number of process activations?

2. Rewrite the architecture of the controller model's architecture to eliminate the conditional signal assignment statements on lines 19 through 21. Instead, use signal assignment statements within the process.

# Packaging Declarations



## ❑ Specification

We are almost ready to build our multiplier. But first, another VHDL capability, one that greatly aids the description of models comprising many separate pieces, must be described.

The page opposite presents a diagram of the hierarchy of the multiplier. At the top is the entity "Mult8" and its associated architecture "IterativeAdd." It will (as we will see in a few pages) incorporate two copies of the shifter (entity "ShiftN" and its associated architecture "Behavior"), and one copy each of the eight-bit latch (entity "Latch8" and its associated architecture "Structure"), the eight-bit adder (entity "Adder8" and its associated architecture "Structure"), and the controller (entity "Controller" and its associated architecture "FSM"). The eight-bit latch in turn incorporates eight copies of the D flip-flop (entity "DFF" and its associated architecture "Dataflow"); similarly, the eight-bit adder incorporates eight copies of the one-bit full adder (entity "FullAdder" and its associated architecture "Equations").

A commonly occurrence in such models is the need to share declarations between the various components in the model. While it is sometimes possible to create multiple copies of the needed declarations, even when it is possible it is inconvenient and leads to great configuration management problems.

Fortunately, VHDL provides the *package*, a construct that allows a single set of declarations to be shared among many models.

```
 1  ------------------------------------------------
 2
 3  package Components is
 4
 5    component Adder8
 6      port (A, B:  in     Bit_Vector(7 downto 0);
 7            Cin:   in     Bit;
 8            Cout:  out    Bit;
 9            Sum:   out    Bit_Vector(7 downto 0));
10    end component;
11
12    component Latch8
13      port (D:     in     Bit_Vector(7 downto 0);
14            Clk:   in     Bit;
15            Pre:   in     Bit;
16            Clr:   in     Bit;
17            Q:     out    Bit_Vector(7 downto 0));
18    end component;
19
20    component ShiftN
21      port (CLK:   in     Bit;
22            CLR:   in     Bit;
23            LD:    in     Bit;
24            SH:    in     Bit;
25            DIR:   in     Bit;
26            D:     in     Bit_Vector;
27            Q:     out    Bit_Vector
28           );
29    end component;
30
31    component Controller
32      port (STB:   in     Bit;
33            CLK:   in     Bit;
34            LSB:   in     Bit;
35            Stop:  in     Bit;
36            Init:  out    Bit;
37            Shift: out    Bit;
38            Add:   out    Bit;
39            Done:  out    Bit);
40    end component;
41
42  end Components;
43
44  ------------------------------------------------
```

## ❑ Implementation—Components

Here is a VHDL *package declaration*, named "Components." Between the reserved words **is**, on line 3, and **end**, on line 41, are the component declarations that we will need in order to build the multiplier. We will shortly see how to incorporate this package in the multiplier model.

This package has no test bench or simulation results, for there is nothing here to execute. However, the package must be compiled just as any other VHDL file. So, prior to continuing, compile the package Components by executing the following commands:

```
% cp $CLSI/examples/tutorial/components.vhdl
% compile components
```

```
 1  ------------------------------------------------
 2
 3  package Utils is
 4
 5    procedure Clock (signal C: out Bit; HT, LT: Time);
 6
 7    function Convert (N,L: Natural) return Bit_Vector;
 8    function Convert (B: Bit_Vector) return Natural;
 9
10  end Utils;
11
12  ------------------------------------------------
13
14  package body Utils is
15
16    procedure Clock (signal C: out Bit; HT, LT: Time) is
17    begin
18      loop
19        C <= '1' after LT, '0' after LT + HT;
20        wait for LT + HT;
21      end loop;
22    end;
23
24    function Convert (N,L: Natural) return Bit_Vector is
25      variable Temp: Bit_Vector(L-1 downto 0);
26      variable Value: Natural := N;
27    begin
28      for i in Temp'Right to Temp'Left loop
29        Temp(i) := Bit'Val(Value mod 2);
30        Value := Value / 2;
31      end loop;
32      return Temp;
33    end;
34
35    function Convert (B: Bit_Vector) return Natural is
36      variable Temp: Bit_Vector(B'Length-1 downto 0) := B;
37      variable Value: Natural := 0;
38    begin
39      for i in Temp'Right to Temp'Left loop
40        if Temp(i) = '1' then
41          Value := Value + (2 ** i);
42        end if;
43      end loop;
44      return Value;
45    end;
46
47  end Utils;
48
49  ------------------------------------------------
```

## ❑ Implementation—Utils

Here is another package, containing a number of subprograms. Unlike the previous package, this package comprises two pieces: a *package declaration* and a *package body*.

Like in the previous package, this package has a package declaration (lines 3 through 10). The package declaration describes those parts of the package that are to be accessible from outside the package. In this declaration we have a *procedure declaration* and two *function declarations*. (Procedures and functions are collectively called *subprograms*; a *subprogram declaration* is the declaration of either a function or a procedure.)

The procedure declaration (line 5) declares a procedure named "Clock." Clock can be called a number of times, to generate a number of clocks, each with a potentially different cycle time and duty cycle. It has three *parameters*. The first, "C", which must be a signal of type bit, is driven by the procedure as the clock. The final two parameters, "HT" and "LT", are time values which respectively represent the time that the clock is high and low. (The cycle time of the clock is HT + LT; the duty cycle is HT / (HT + LT).) This procedure is intended to be called concurrently.

Lines 7 and 8 contain two function declarations. These functions are to be used to convert between bit vectors and integers. Both are called "Convert"; VHDL allows subprograms to be *overloaded* under certain circumstances.

The "Convert" on line 7 accepts two parameters, both of type "Integer" (recall that Natural is a predefined subtype of Integer), and returns a bit vector value—this function converts the non-negative integer, N to a bit vector of length L whose bits represent N in unsigned magnitude notation. The "Convert" on line 8 accepts a single, bit vector parameter and returns an Integer value, constrained to the subtype Natural—this function converts the bit vector B to a non-negative integer assuming unsigned magnitude notation.

Any one of these characteristics serves to distinguish the overloaded subprograms: the return type (a procedure is considered to have a return type distinct from any function), the number of parameters, and the types (not the subtypes of the parameters). As long as the VHDL compiler can distinguish the subprograms from each other by at least one of these characteristics, overloading is allowed.

Because the package declaration contains subprogram declarations, the package requires a package body. The package body (lines 14 through 47), contains *subprogram bodies* that implement the subprograms whose declarations are contained in the package declaration—it may also contain other declarations that are to be used internally to the package body. Each subprogram declared in the package declaration must have a corresponding body in the package body.

Lines 16 through 22 give one such body, that for the procedure "Clock." There are several important things to note about the body. The first is that the procedure declaration of line 5 must be repeated exactly (up to, but not including the terminal semicolon) as the first part of the body (line 16). This repetition ensures that the VHDL compiler knows which subprogram interface is being completed by this body.

```
 1  ------------------------------------------------
 2
 3  package Utils is
 4
 5    procedure Clock (signal C: out Bit; HT, LT: Time);
 6
 7    function Convert (N,L: Natural) return Bit_Vector;
 8    function Convert (B: Bit_Vector) return Natural;
 9
10  end Utils;
11
12  ------------------------------------------------
13
14  package body Utils is
15
16    procedure Clock (signal C: out Bit; HT, LT: Time) is
17    begin
18      loop
19        C <= '1' after LT, '0' after LT + HT;
20        wait for LT + HT;
21      end loop;
22    end;
23
24    function Convert (N,L: Natural) return Bit_Vector is
25      variable Temp: Bit_Vector(L-1 downto 0);
26      variable Value: Natural := N;
27    begin
28      for i in Temp'Right to Temp'Left loop
29        Temp(i) := Bit'Val(Value mod 2);
30        Value := Value / 2;
31      end loop;
32      return Temp;
33    end;
34
35    function Convert (B: Bit_Vector) return Natural is
36      variable Temp: Bit_Vector(B'Length-1 downto 0) := B;
37      variable Value: Natural := 0;
38    begin
39      for i in Temp'Right to Temp'Left loop
40        if Temp(i) = '1' then
41          Value := Value + (2 ** i);
42        end if;
43      end loop;
44      return Value;
45    end;
46
47  end Utils;
48
49  ------------------------------------------------
```

Another important feature is that the statements inside of the body (lines 18 through 21) constitute an indefinite loop. Recall that this procedure is intended to be called concurrently. Such procedures are "called" once and must repeatedly execute their bodies; should they ever return to the caller, they will never be called again.

The statements inside of the loop implement the clock. The first, on line 19, drives the clock signal high after LT, then back low after LT + HT. The wait statement on line 20 ensures that the loop pauses long enough for these changes take place on the clock. The enclosing loop then causes the statements to be re-executed.

Lines 24 through 33 and 35 through 45 provide the bodies for the two "Convert" functions. Examining the function returning a bit vector first, we see that it declares a bit vector of the correct length on line 25 (by using the function's parameter, "L"), and makes a copy of N on line 26. (We need to decrement N as we proceed, and VHDL will not let us modify N.) The loop on lines 28 through 31 runs through the bit positions of Temp in right to left order (by using the predefined attributes of arrays "Right" and "Left"). For each bit, we see if the current remaining value left to convert is even or odd. If it is even, the current bit should be '0', otherwise it should be '1'. We generate the '0' or '1', by using a new predefined attribute of types, "Val." "Val" works as follows: the expression "Value mod 2" is zero if Value is even, otherwise the expression is one. "Bit'Val(0)" returns the value of Bit whose *position number* is zero, namely '0', while "Bit'Val(1)" returns the value of Bit whose position number is one, or '1'. (The position number of a *scalar type* is its distance from the left-most value of the type. Enumeration types and numeric types are scalar types.)

Once the loop is finished, it exits and the *return statement* on line 32 executes. Its execution causes the function to return to the caller, returning the bit vector value stored in "Temp" as the result of calling the function. (A function may have more than one return statement, but they all must return values of the correct type and the function must return by means of a return statement. A procedure may, but need not return via a return statement; a return statement used in a procedure must not return a value.)

The "Convert" function, whose body begins on line 35, works in slightly different fashion. It first makes a copy of B in the Temp variable (line 36), so that we know both the end points and the direction of its index. (Recall that a bit vector may have, as the endpoints of its index, any non-negative integers, and that the range itself may be either ascending or descending.) We also create an accumulator variable on line 37, whose initial value is zero.

The statements of this function examine the bits of Temp, again in right-to-left order. For each bit of Temp, if the bit is '1', we add $2^i$ to the accumulator, where $i$ is the distance of the current bit from the right-most bit. When the loop is exhausted, Value contains the desired return value, which we return on line 44.

```
50  ------------------------------------------------
51
52  entity Test_Utils is end;
53
54  ------------------------------------------------
55
56  use work.Utils.all;
57  architecture Driver of Test_Utils is
58
59    signal N: Natural;
60    signal B: Bit_Vector(5 downto 0);
61    signal C: Bit;
62    use work.Utils.all;
63
64  begin
65
66    CLK: Clock(C, 10 ns, 20 ns);
67
68    process
69      use work.Utils.all;
70    begin
71      for i in 0 to 31 loop
72        B <= Convert(i, B'Length) after 10 ns;
73        wait for 10 ns;
74      end loop;
75      wait;
76    end process;
77
78    N <= Convert(B) after 1 ns;
79
80  end;
81
82  ------------------------------------------------
```

## ❑ Verification

The test bench for the utilities package differs in a number of ways from the test benches for the earlier examples. Since the utilities package is a package, not an entity–architecture pair, we do not declare and instantiate a component in order to create a unit under test; instead, we *use* the package with a *use clause*, then call the functions and procedures in the test bench.

The first step, the use clause, is shown three times in this example, on lines 56, 62, and 69. Any two of these use clauses are redundant, but are shown merely to display the different places where a use clause can be placed.

The use clause "**use** work.Utils.**all**;" is interpreted as follows: In library WORK, locate a previously compiled package called "Utils". "Import" all declarations in the package's declaration (but not those of its body). (This use clause causes references to "Clock" and "Convert" in the test bench will refer to the subprograms declared in Utils.) Thus, use clauses act somewhat like the "#include …" directive of C.

Line 66 tests the "Clock" procedure with a *concurrent procedure call*. (Recall that the statements in an architecture execute concurrently. Thus, this procedure executes concurrently with the process on lines 68 through 76 and with the signal assignment statement on line 78—examine the simulation output on the next page to satisfy yourself that this is so.)

Recall that the body of "Clock" is an indefinite loop. Thus, we cannot simulate it until the unit under test becomes stable—the clock, by definition, never stabilizes! We need a "break after a specified time has elapsed" command (which we will introduce on the next page).

But first, let us examine the tests for the two overloaded functions in the Utils package. The "Convert" function returning a bit vector is tested on line 72. Every ten nanoseconds it is fed an integer in the range $0 \le i \le 31$. The result is put in the signal B. Every time B changes, the signal assignment on line 78 calls the Convert returning an integer; the returned integer is placed on the signal N one nanosecond later.

How does the VHDL compiler understand which "Convert" to call? Recall the distinguishing characteristics of overloaded subprograms mentioned previously. The Convert called on line 72 has two parameters supplied to it, both integers. Moreover, a bit vector return is needed, since B is a bit vector. There is only one Convert function that matches in number and types of parameters and in return type; that one is the one called.

Similarly, the Convert on line 78 is supplied with a single, bit vector parameter and an integer return is needed. Again, only one Convert matches this specification. (It is an error if no functions with the given name match all the requirements; it is also an error if more than one function with the given name match all the requirements.)

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
       1   : /test_utils(driver).n
   2 - 7   : /test_utils(driver).b
       8   : /test_utils(driver).c

                                    1    2    3    4    5    6    7    8
      Time(fs) + Cycle              n b(5) b(4) b(3) b(2) b(1) b(0)    c
----------------------  ------------ ---- ---- ---- ---- ---- ---- ----
                  0+ 0:              0  '0'  '0'  '0'  '0'  '0'  '0'  '0'
            1000000+ 0: *            0  '0'  '0'  '0'  '0'  '0'  '0'  '0'
           10000000+ 0:              0 *'0' *'0' *'0' *'0' *'0' *'0'  '0'
           20000000+ 0:              0 *'0' *'0' *'0' *'0' *'0' *'1' *'1'
           21000000+ 0: *            1  '0'  '0'  '0'  '0'  '0'  '1'  '1'
           30000000+ 0:              1 *'0' *'0' *'0' *'0' *'1' *'0' *'0'
           31000000+ 0: *            2  '0'  '0'  '0'  '0'  '1'  '0'  '0'
           40000000+ 0:              2 *'0' *'0' *'0' *'0' *'1' *'1'  '0'
           41000000+ 0: *            3  '0'  '0'  '0'  '0'  '1'  '1'  '0'
           50000000+ 0:              3 *'0' *'0' *'0' *'1' *'0' *'0' *'1'
           51000000+ 0: *            4  '0'  '0'  '0'  '1'  '0'  '0'  '1'
           60000000+ 0:              4 *'0' *'0' *'0' *'1' *'0' *'1' *'0'
           61000000+ 0: *            5  '0'  '0'  '0'  '1'  '0'  '1'  '0'
           70000000+ 0:              5 *'0' *'0' *'0' *'1' *'1' *'0'  '0'
           71000000+ 0: *            6  '0'  '0'  '0'  '1'  '1'  '0'  '0'
           80000000+ 0:              6 *'0' *'0' *'0' *'1' *'1' *'1' *'1'
           81000000+ 0: *            7  '0'  '0'  '0'  '1'  '1'  '1'  '1'
           90000000+ 0:              7 *'0' *'0' *'1' *'0' *'0' *'0' *'0'
           91000000+ 0: *            8  '0'  '0'  '1'  '0'  '0'  '0'  '0'
          100000000+ 0:              8 *'0' *'0' *'1' *'0' *'0' *'1'  '0'
          101000000+ 0: *            9  '0'  '0'  '1'  '0'  '0'  '1'  '0'
          110000000+ 0:              9 *'0' *'0' *'1' *'0' *'1' *'0' *'1'
          111000000+ 0: *           10  '0'  '0'  '1'  '0'  '1'  '0'  '1'
          120000000+ 0:             10 *'0' *'0' *'1' *'0' *'1' *'1' *'0'
          121000000+ 0: *           11  '0'  '0'  '1'  '0'  '1'  '1'  '0'
          130000000+ 0:             11 *'0' *'0' *'1' *'1' *'0' *'0'  '0'
          131000000+ 0: *           12  '0'  '0'  '1'  '1'  '0'  '0'  '0'
          140000000+ 0:             12 *'0' *'0' *'1' *'1' *'0' *'1' *'1'
          141000000+ 0: *           13  '0'  '0'  '1'  '1'  '0'  '1'  '1'
          150000000+ 0:             13 *'0' *'0' *'1' *'1' *'1' *'0' *'0'
          151000000+ 0: *           14  '0'  '0'  '1'  '1'  '1'  '0'  '0'
          160000000+ 0:             14 *'0' *'0' *'1' *'1' *'1' *'1'  '0'
          161000000+ 0: *           15  '0'  '0'  '1'  '1'  '1'  '1'  '0'
          170000000+ 0:             15 *'0' *'1' *'0' *'0' *'0' *'0' *'1'
          171000000+ 0: *           16  '0'  '1'  '0'  '0'  '0'  '0'  '1'
          180000000+ 0:             16 *'0' *'1' *'0' *'0' *'0' *'1' *'0'
          181000000+ 0: *           17  '0'  '1'  '0'  '0'  '0'  '1'  '0'
          190000000+ 0:             17 *'0' *'1' *'0' *'0' *'1' *'0'  '0'
          191000000+ 0: *           18  '0'  '1'  '0'  '0'  '1'  '0'  '0'
          200000000+ 0:             18 *'0' *'1' *'0' *'0' *'1' *'1' *'1'
          201000000+ 0: *           19  '0'  '1'  '0'  '0'  '1'  '1'  '1'
          210000000+ 0:             19 *'0' *'1' *'0' *'1' *'0' *'0' *'0'
          211000000+ 0: *           20  '0'  '1'  '0'  '1'  '0'  '0'  '0'
          220000000+ 0:             20 *'0' *'1' *'0' *'1' *'0' *'1'  '0'
          221000000+ 0: *           21  '0'  '1'  '0'  '1'  '0'  '1'  '0'
          230000000+ 0:             21 *'0' *'1' *'0' *'1' *'1' *'0' *'1'
          231000000+ 0: *           22  '0'  '1'  '0'  '1'  '1'  '0'  '1'
          240000000+ 0:             22 *'0' *'1' *'0' *'1' *'1' *'1' *'0'
          241000000+ 0: *           23  '0'  '1'  '0'  '1'  '1'  '1'  '0'
          250000000+ 0:             23 *'0' *'1' *'1' *'0' *'0' *'0'  '0'
          251000000+ 0: *           24  '0'  '1'  '1'  '0'  '0'  '0'  '0'
          260000000+ 0:             24 *'0' *'1' *'1' *'0' *'0' *'1' *'1'
```

## ❑ Running the Example

We must simulate the utilities package with care. Recall that the clock procedure will run indefinitely, so a naive batch simulation, or an interactive simulation without sufficient breakpoints, will not terminate. By inspecting the test bench, we can determine that the process that tests the "Convert" routines requires 321 ns to complete, which is adequate time in which to test the clock. Thus, we need to run the simulation for at least 321 ns, at which point we wish to regain control of the simulation so that we can terminate it gracefully.

The simplest way is to use the *break time* command. This command is available when simulating in interactive mode. Use the following instructions, and note the effect of the commands you haven't used before:

```
% cp $CLSI/examples/tutorial/utils.vhdl .
% compile utils -debug
% build test_utils
% simulate test_utils -interactive -trace
Command> elaborate
Command> break time 321 ns
Command> show break
Command> go
Command> cancel break 1
Command> show break
Command> quit
%
% format test_utils
```

Note that the results opposite have been truncated at 260 ns (260000000 fs) in order to fit the page.

## ❑ Additional Exercises

1. Add a parameter, StartTime, to the Clock procedure. This parameter, of type time, specifies how long the clock remains quiescent before beginning to cycle.

2. Add another parameter, StopTime, to the Clock procedure, that specifies the amount of time the clock runs before becoming quiescent. You may find the predefined function "Now", which returns the current time, useful. "Now" is declared as follows:

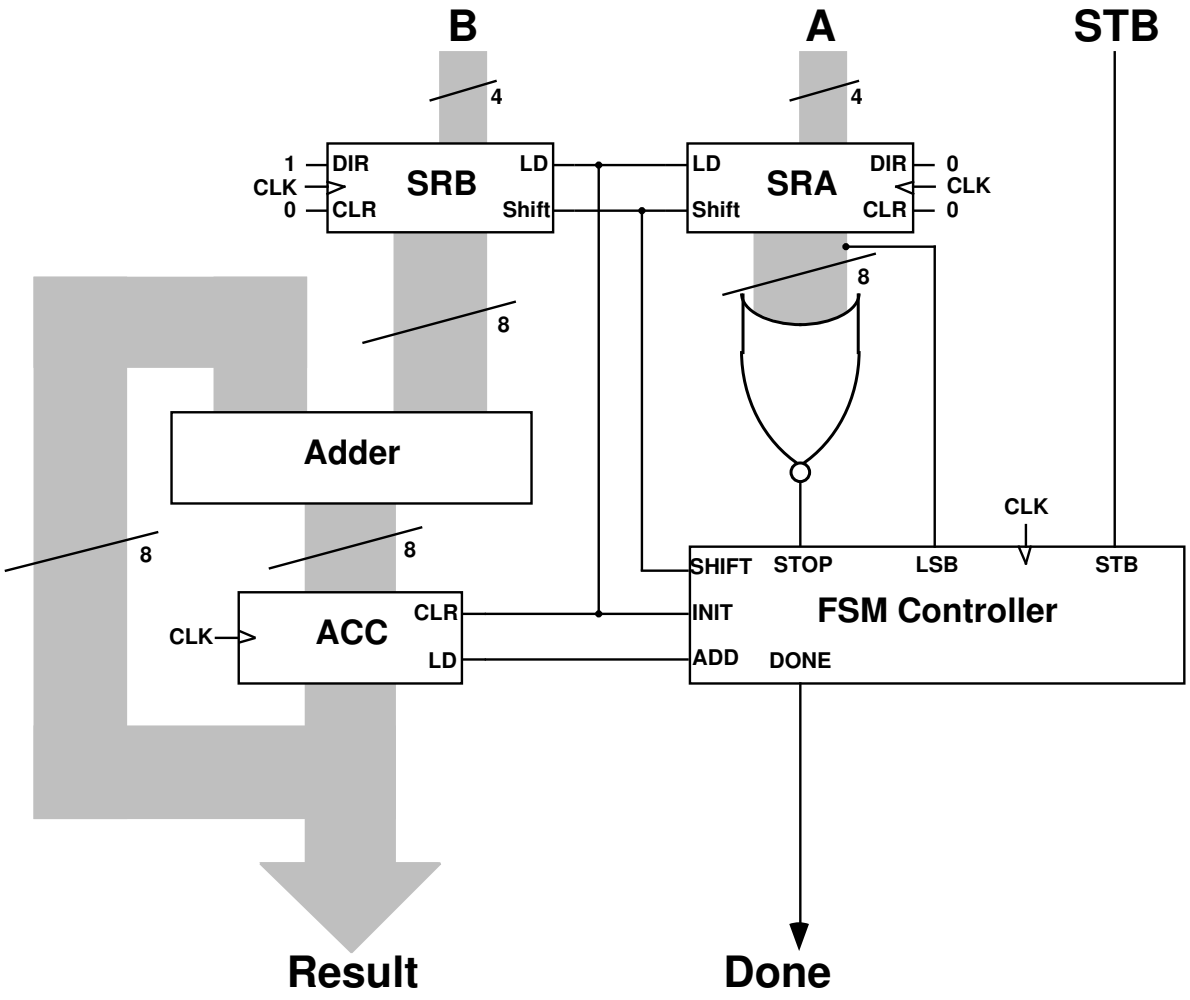
```
function Now return Time;
```

   Use an assertion statement to ensure that StartTime ≤ StopTime.

3. Modify the Convert functions to use signed-magnitude representation for integers. Test them using the range $-16 \le i \le 16$.

4. Modify the Convert functions to use twos-complement integers. Test them using the range $-15 \le i \le 16$.

## Mixed Modeling—A Simple Multiplier



## ❑ Specification

At long last, the multiplier. The system inputs are the two four-bit multiplicands, A and B, a clock signal, and a strobe signal to start the machine. The system outputs are the eight-bit signal "Result" and a "Done" bit, indicating that the result is valid.

When the strobe signal STB is received, the controller resets and generates an INIT pulse, which clears the accumulator and latches the A and B inputs into the shift registers. SRA is wired to shift to the right, and SRB is wired to shift to the left, so that when the SHIFT signal is received, they respectively implement A := A div 2 and B := 2 * B; when the ADD signal is received the current accumulator contents are added to the current contents of SRB and stored back in the accumulator. These signals are generated by the controller responding to the LSB bit, taken directly from the least-significant bit of SRA, and by the STOP bit, which is the NOR of all eight bits of SRA's output.

```
 1  -------------------------------------------------
 2
 3  entity Mult8 is
 4    port (A,B:    in      Bit_Vector(3 downto 0);
 5          STB:    in      Bit;
 6          CLK:    in      Bit;
 7          Reset:  in      Bit;
 8          Result: out     Bit_Vector(7 downto 0);
 9          Done:   out     Bit);
10  end Mult8;
11
12  -------------------------------------------------
13
14  architecture IterativeAdd of Mult8 is
15
16    use Work.Components.all;
17
18    signal SRAout, SRBout:   Bit_Vector(7 downto 0);
19    signal ADDout, ACCout:   Bit_Vector(7 downto 0);
20    signal Init, Shift, Add, Stop: Bit := '0';
21
22    signal High: Bit := '1';
23    signal Low:  Bit := '0';
24    signal OFL:  Bit;
25
26    signal ACCclk, ACCclr: Bit;     -- filtered ACC inputs
27
28  begin
29
30    ACCclr <= not (Init or Reset);  -- ACC clear is active low
31    ACCclk <= CLK and Add;          -- Add enables CLK to ACC
32    Result  <= ACCout;              -- ACC output drives Result
33
34    SRA: ShiftN port map
35           (CLK => CLK,
36            CLR => Low,   -- wired low; not used
37            LD  => Init,
38            SH  => Shift,
39            DIR => Low,   -- wired low for right shift
40            D   => A,
41            Q   => SRAout);
42
43    SRB: ShiftN port map
44           (CLK => CLK,
45            CLR => Low,   -- wired low; not used
46            LD  => Init,
47            SH  => Shift,
48            DIR => High,  -- wired high for left shift
49            D   => B,
50            Q   => SRBout);
51
52    ALU: Adder8 port map
53           (ACCout, SRBout, Cin=>Low, Cout=>OFL, Sum=>ADDout);
54
55    ACC: Latch8 port map
56           (ADDout, Q=>ACCout, Pre=>High, Clk=>ACCclk, Clr=>ACCclr);
57
58    CHK:  Stop <= not (SRAout(7) or SRAout(6) or SRAout(5) or SRAout(4) or
59                       SRAout(3) or SRAout(2) or SRAout(1) or SRAout(0));
60
61    FSM: Controller port map
62           (STB, CLK, SRAout(0), Stop, Init, Shift, Add, Done);
63
64  end;
65
66  -------------------------------------------------
```

## ❑ Implementation

The multiplier model is shown at left. Note how we solved the problem of feeding back the "Result" signal. "Result" is described (on line 8) as a port of mode **out**; we will use an internal signal ("ACCout"— line 19 declares it and line 32 drives "Result" with it) to buffer its value.

The architecture introduces only a few new features of VHDL. Notice that it is a mixed, structure–dataflow description, as it uses signal assignment statements (on lines 31, 31, 32, and 58–59) to describe part of the circuit, and component instantiation statements to describe the rest. One of the most powerful features of VHDL is this ability to intermix different description styles within the same architecture.

The component declarations for these instances are found in the "Components" package; the use clause on line 16 makes these available for use in the architecture.

The instances themselves introduce a new feature. The instantiation of the controller (lines 61 and 62) it associates signals with the component's ports using *positional association*, as we have up to this point used in all examples.

In positional association, the first signal named in the port map is matched up with the first port of the component, the second signal with the second port, etc. Thus, for the controller, the following table of port–signal associations may be built:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Controller's Port:** | STB | CLK | LSB | Stop | Init | Shift | Add | Done |
| **Multiplier's Signal:** | STB | CLK | SRAout (0) | Stop | Init | Shift | Add | Done |

Note that this information is represented implicitly by the position of the multiplier's signal, called the *actual*, in the port map.

It is also possible to specify these associations explicitly, using *named association*. Examine, for example, the instance of the shift register on lines 34 through 41. Each association in the port map is of the form "*component_port => multiplier_signal*". The component port is called the *formal*; the multiplier signal is once again the actual in the association.

There are several reasons why named association would be used; one is to explicitly document the associations. Another is to eliminate the need to associate the formals with the actuals in the order in which the formals are declared. Since the formals in the associations are explicitly represented, the order of association is not relevant (as it is in positional association).

It is also possible to combine named and positional association in the same *association list* (a port map is one such association list). Examine the port map for the adder (lines 52 and 53), for example. The first two formals are positionally associated; the remaining three are associated using the named notation. *All positional association (if any) in an association list must precede any use of named association.* Otherwise, the VHDL compiler cannot understand what actuals are being associated with which formals.

```
 67  -------------------------------------------------
 68
 69  entity Test_Mult8 is end;
 70
 71  -------------------------------------------------
 72
 73  architecture Driver of Test_Mult8 is
 74
 75    component Mult8
 76      port (A,B:    in     Bit_Vector(3 downto 0);
 77            STB:    in     Bit;
 78            CLK:    in     Bit;
 79            Reset:  in     Bit;
 80            Result: out    Bit_Vector(7 downto 0);
 81            Done:   out    Bit);
 82    end component;
 83
 84    signal A, B: Bit_Vector(3 downto 0);
 85    signal STB, Done: Bit := '0';
 86    signal CLK: Bit;
 87    signal Reset: Bit;
 88    signal Result: Bit_Vector(7 downto 0);
 89
 90    signal DisplayA, DisplayB, DisplayResult: Natural;
 91
 92    use Work.Utils.all;
 93
 94  begin
 95
 96    C:   Clock(CLK, 10 ns, 10 ns);
 97
 98    UUT: Mult8 port map (A, B, STB, CLK, Reset, Result, Done);
 99
100    Reset  <= '1', '0' after 1 ns;  -- to reset Mult8 at start
101
102    Stimulus:
103      process
104      begin
105         for i in 1 to 3 loop
106          for j in 4 to 7 loop
107            DisplayA <= i;
108            DisplayB <= j;
109            A <= Convert(i, A'Length);
110            B <= Convert(j, B'Length);
111            wait until CLK'Event and CLK='1';
112            STB <= '1', '0' after 20 ns;
113            wait until Done = '1';
114            DisplayResult <= Convert(Result);
115            wait until CLK'Event and CLK='1';
116          end loop;
117        end loop;
118        wait;
119      end process;
120
121  end;
122
123  -------------------------------------------------
```

## ❑ Verification

Here is a test bench for the multiplier. We use the Clock procedure from the utilities package as a concurrent procedure (line 96) to generate a clock for the multiplier and test bench. Because this clock will run indefinitely, we must use the *break time* command when simulating this test bench.

The stimulus generator (lines 102 through 119) uses two nested loops to test the multiplier with all values where A is taken from {1, 2, 3} and B is taken from {4, 5, 6, 7}.

Within the inner loop, it first applies the values to be multiplied to the signals "DisplayA" and "DisplayB" so that they may be displayed as integers (lines 107 and 108), then converts them to bit vectors for multiplication using the appropriate Convert procedure from the utilities package (lines 109 and 110); these statements also assign these values to the inputs of the multiplier.

The wait statement on line 111 then waits for a rising edge on clock. (The until condition is the most general form of event-sensitive expression, although the term "CLK'Event" in the wait statement is redundant in this case.) The STB signal is then strobed (line 112) to initiate multiplication.

The next wait statement (line 113) causes the test bench to wait for the multiplier to indicate that it is finished. The result is then converted to an integer and assigned to the signal "DisplayResult" so that it also may be viewed as an integer (line 114). Finally, the wait statement on line 115 pauses until the next rising edge on clock in order to allow the results to be displayed.

```
VHDL Simulation System Version 1.5.0.2


Signal       Signal
Number       Path Name
----------   ------------------------------------------------------------------
       21  : /test_mult8(driver).displaya
       22  : /test_mult8(driver).displayb
       23  : /test_mult8(driver).displayresult

                                  21           22            23
      Time(fs) + Cycle       displaya     displayb displayresult
----------------------  ------------ ------------ -------------
                   0+ 0:            0            0             0
                   0+ 1: *          1 *          4             0
...
            97000000+ 2:            1            4 *           4
...
           110000000+ 1: *          1 *          5             4
...
           217000000+ 2:            1            5 *           5
...
           230000000+ 1: *          1 *          6             5
...
           337000000+ 2:            1            6 *           6
...
           350000000+ 1: *          1 *          7             6
...
           457000000+ 2:            1            7 *           7
...
           470000000+ 1: *          2 *          4             7
...
           617000000+ 2:            2            4 *           8
...
           630000000+ 1: *          2 *          5             8
...
           777000000+ 2:            2            5 *          10
...
           790000000+ 1: *          2 *          6            10
...
           937000000+ 2:            2            6 *          12
...
           950000000+ 1: *          2 *          7            12
...
          1097000000+ 2:            2            7 *          14
...
          1110000000+ 1: *          3 *          4            14
...
          1277000000+ 2:            3            4 *          12
...
          1290000000+ 1: *          3 *          5            12
...
          1457000000+ 2:            3            5 *          15
...
          1470000000+ 1: *          3 *          6            15
...
          1637000000+ 2:            3            6 *          18
...
          1650000000+ 1: *          3 *          7            18
...
          1817000000+ 2:            3            7 *          21
```

## ❑ Running the Example

Since the test bench for the multiplier uses the Clock procedure defined in the utilities package, the *break time* command should be used when simulating. Also, to keep the output manageable, only the "DisplayA", DisplayB", and "DisplayResult" signals are formatted. Try the following command sequence for this simulation:

```
% cp $CLSI/examples/tutorial/mult8.vhdl .
% compile mult8 -debug
% build test_mult8
% simulate test_mult8 -interactive -trace
Command> break time 2000 ns
Command> go
Command> quit
```

When simulation completes, format the resulting trace file to look like the opposite page.

## ❑ Additional Exercises

1. Add code to the test bench to verify the correctness of the results. You may wish to use VHDL's multiplication operator "*" in this exercise.

2. Modify the test bench to check the full range of the multiplier: $0 \leq A, B \leq 7$. Characterize the multiplication time as a function of the input values.

3. Using the results of exercise 2, add additional circuitry to the multiplier to "steer" the inputs so as to minimize the multiplication time.

## The Full Power of VHDL

You now know enough VHDL to model systems of moderate complexity, but VHDL encompasses many more features that we have presented in this brief tutorial. Among these features are the following:

| | |
|---|---|
| access types | for high-level modeling using dynamically allocated data structures; |
| block statements | for modeling explicit hierarchy; |
| configurations | for controlling the configuration of models; |
| file input-output | for data collection, memory initialization and inter-tool communication; |
| generics | for modeling environmental and parametric information; |
| guarded signals | for the high-level modeling of registers, busses, and latches; |
| physical types | for modeling measurable quantities; |
| record types | for modeling heterogeneous data; |
| resolution functions | for modeling multiply driven signals; and |
| transport delay | for modeling wiring delay; |

In the directory $CLSI/examples/mpu you will find an extended example of the use of VHDL – a simple microprocessor, along with a program you can execute on the simulated machine. The "commands" file contains the commands necessary to compile the example. Build and simulate the model "test_mpu". You can now run the simulation. You will find it rewarding to study the source code of this example to learn more about VHDL.

# PART 2: VHDL REFERENCE GUIDE

# CHAPTER 1
# SIMULATOR COMMAND SUMMARY

The Scout VHDL Simulation System (VSS) is a full IEEE standard VHDL 1076 simulation system. The VSS consists of a VHDL analyzer and design library system, a code generator and simulation model builder, a batch/interactive simulator, and a signal trace formatter.

This chapter describes each of the commands provided by the VSS for developing a simulation model, for simulating the model, for formatting and displaying the results of simulation, and for examining and manipulating the contents of design libraries.

Commands are organized into logical groups in the order in which they are typically used. A template for each command is presented in *italics*. Items in angle brackets (e.g. "*<number>*") indicate parameters that should be replaced with actual values. Items in square brackets (e.g. "*[ -interactive ]*") are optional. A list of items in braces separated by a vertical bar (e.g." *{ off | on }*") indicates that only one of the list of items may be used. An ellipsis (e.g. "...") indicate that the immediately preceding parameter may be repeated zero or more times.

## Top-Level Commands

The following commands are available for building simulation models, simulating them, and working with design libraries.

### ❑ Library Commands

*setenv dls_<library_name> <directory_specification>*

*setenv WORK dls_<library_name>*

Design libraries are implemented in the VSS as directories in the underlying operating system. A directory is identified as a design library by one permanent library name and any number of alternative library names.

A permanent library name is implemented as a shell environment variable beginning with the string "dls_". For example, the permanent library name for the library "ieee" is "dls_ieee". The value of the permanent library name is the path name of a directory. The directory contains library units, object code, and configuration data files used for model building. Once you have stored design units in the library the permanent name cannot be changed.

You may define any number of alternate names for a library. Unlike a permanent name, an alternate name may be reassigned to a different library at any time. The name of the environment variable representing an alternate library must not begin with "dls_". The value of the environment variable must be a permanent library

name. You may refer to a library from within a VHDL design unit using either the <library_name> part of its permanent name or an alternate name.

The first command illustrated tells VSS that the indicated directory corresponds to the VHDL library <library_name>. The second command tells VSS to use the library as the default working library for all compilation and model building commands.

*list libraries*

Lists the library name and associated directory for each available design library.

*list units [ <library_name> ]*

Lists the VHDL design units that exist in the specified library. The default library is "WORK". Each entry in the list represents a library unit, a configuration data file, or an object code file corresponding to a VHDL design unit from a compiled VHDL source file.

Top-level entities (those with no generics and no ports) are marked as such in the list. (See the *build* command for information about building simulation models for top-level entities.)

Library units that depend on other library units that have been modified since the dependency was created are *out-of-date* library units. They must be recompiled from the original source VHDL before they can be used. Such library units are marked as out-of-date in the list. (See the *compile* command for information about compiling VHDL source files. See the *list libraries* command to display the names of the available libraries.)

*list models [<library_name>]*

Lists the simulation models that have been built in the specified design library. The default library is "WORK". (See the *simulate* command for information about executing a simulation model. See the *build* command for building simulation models. See the *list libraries* command to display the names of the available libraries.)

*delete unit <library_unit>*

Deletes a library unit, its configuration data, and its object code from library "WORK". If the library unit is a top-level entity and there is a simulation model corresponding to that entity in the library, that simulation model is unaffected.

*delete model <model_name>*

Deletes a simulation model from library "WORK". Library units that were used to build the simulation model are unaffected.

## ❑ Compile Command

*compile <vhdl_source_file> [ -[no]debug ] [ -analyze ] [ -generate ] [ -optimize ] [ -norange_checks ] [ -build …]*

Reads the VHDL source files specified and generates a library unit, configuration data and object code for each successfully analyzed design unit within the source files. The resulting library units, configuration data and object files are stored within the design library "WORK". (See the *list units* command for displaying the library units within library "WORK".)

The compiler produces non-debuggable code by default. If you intend to simulate in interactive mode, use the -debug switch. Compilation proceeds in three steps; first analysis, then code generation, finally object file generation. The -analyze switch suppresses the last two steps; the -generate switch suppresses the first step. The -optimize and -norange_checks switches speed simulator execution at the expense, in the first case, of compilation time; and in the second case, of run-time checking of array references. The -build switch is followed by model building instructions in the form of the *build* command (below).

The VHDL source file is unaffected by this command. However, the object files created contain references to the VHDL source file for source-level debugging support, so it is advisable to put the VHDL source file in a permanent place within the file system before executing the *compile* command. For example, VHDL source files can be stored in a directory that is also used as the design library for the corresponding library units.

## ❑ Build Command

*build <top_level_unit> [ -name <model_name> ]*

Builds a simulation model for the specified top-level design unit from object files in the library "WORK".

The top level unit specification can be the name of an entity, entity-architecture pair, or configuration. If an entity is specified, the entity together with its most-recently-compiled architecture define the top-level design entity of the model. If an entity-architecture pair is specified, the specified entity and architecture define the top-level design entity. An entity-architecture pair is specified with the name in the form: *<entity_name> - <architecture_name>* . If a configuration is selected, the entity and architecture referenced by the configuration define the top-level design entity.

The top-level design entity must be a test bench whose entity declaration contains no generics and no ports. The resulting executable image is stored within the library "WORK". (See the *list models* command for displaying the simulation models within library "WORK".)

The object files required for a given simulation model are determined by starting with the library unit corresponding to the specified top-level design entity and recursively traversing the design hierarchy.

The *-name* switch optionally specifies the name of the resulting simulation model. The default name of the simulation model is the same as the parameter given to specify the top-level design entity.

## ❑ Simulate Command

*simulate <model_name> [ -interactive ] [ -trace ] [ -noflatten_ports ] [ -fast_resolution ] [ -verbose ] [ -command_file <file_name> ] [ -cycle_limit <number>]*

Executes the specified simulation model. (See the *list models* command for listing simulation models available within library "WORK".)

The *-interactive* switch allows the user to monitor the course of simulation through the use of simulation control commands. The user is prompted for a command after the simulation model is loaded and before elaboration begins.

If *-trace* is enabled, all signal activity is traced, and a signal trace file with the name "<model_name>.trace" is created after simulation completes. (See the *format* command for formatting some or all of the signal traces in the trace file.)

The VSS system optimizes a simple hierarchical net by flattening it into a single signal. Simple nets contain only signals and ports with neither resolution functions nor type conversion functions. This optimization can be turned off with the *-noflatten_ports* switch. The *-fast_resolution* switch extends this to resolved signals when it is possible. and also precalculates resolution function values to speed execution.

The *-verbose* switch causes informational message to be issued at each simulation cycle.

The *-command_file* switch specifies the file that contains simulation commands to be executed at the beginning of simulation. If the file name is specified, then that file is used, otherwise the default file name "<model_name>.script" is used. Files can also be executed with the *do* command.

the *-cycle_limit* switch limits execution to the specified number of simulation cycles, thus preventing a runaway execution during batch simulation.

## ❑ Signal Trace Formatting Command

*format <model_name> [ -preview ] [ -nonames ]*

Formats, or provides information about formatting, the signal trace file generated by executing the specified simulation model.

If the *-preview* switch is specified, then the following occurs: first, a format control file (<model_name>.control) is generated if one does not already exist in the current working directory; next, the directives in the format control file are processed to determine the number of signals that would be displayed in the formatted output waveform file and the number of columns required for that display; finally, a message containing that information is written to *stdout*.

If the *-preview* switch is not specified, then a formatted output waveform file is generated from information in the signal trace file (<model_name>.trace). Formatting is performed based upon directives in the format control file (<top_level_entity>.control), if that file is present in the current directory; otherwise, all signals in the signal trace file are formatted. The resulting formatted output waveform file is named "<top_level_entity>.wave".

If the *-nonames* switch is specified, then the output waveform file is generated without signal names in the column headings.

# Interactive Simulation Commands

A variety of subcommands are available for interactively monitoring and controlling the course of simulation.

Several of the commands operate from a point of reference in the design hierarchy. The current point of reference is the "context region". (See the *set region* command to set the context region. See the *examine* command to display the values of objects in the context region.)

## ❑ Simulation Macro-Step Commands

*elaborate*

Causes the model to execute until a breakpoint is encountered or the design hierarchy has been fully elaborated. If the model has already been fully elaborated, the command has no effect.

*initialize*

Causes the model to execute until a breakpoint is encountered or until the design hierarchy has been elaborated and initialized. If the model has already been initialized, the command has no effect.

*go*

Causes the model to execute until a breakpoint is encountered or until the design hierarchy has been elaborated and initialized and simulation has terminated. If simulation has already terminated, the command has no effect.

## ❑ Simulation Micro-Step Commands

*step*

Causes simulation to advance one step (i.e., either one declaration or one statement). If the step involves a procedure call or a function call, then execution will stop at the first line of the called procedure or function. If *step* is issued when a wait statement is about to be executed, then simulation steps to the next breakpoint encountered or to the resumption point of the next process to be executed, whichever comes first.

*next*

Causes simulation to advance one step. The *next* command is similar to *step*, except that *next* executes called procedures or functions without stopping (unless a breakpoint is encountered within the procedure or function).

*step suspend*

Causes simulation to advance until the currently executing process suspends (by executing an implicit or explicit wait statement).

*step cycle*

Causes simulation to advance until all processes scheduled to execute in the current simulation cycle have suspended, and time is ready to advance.

*step process*

Causes simulation to advance to the resumption point of the next process to be executed.

## ❑ Model Information Display Commands

*show hierarchy*

Displays the elaborated design hierarchy contained within the model. If the *show hierarchy* command is issued before elaboration is completed, then only that part of the design hierarchy that has been elaborated so far will be displayed.

*show processes*

Displays a list of the processes making up the "leaf level" of the design hierarchy. If the *show processes* command is issued before elaboration is completed, then only those processes that have been elaborated so far will be displayed.

*show queue*

Displays a list of the drivers with pending transactions and a list of processes waiting to resume no later than a particular time.

*show signals*

Displays a list of the signals declared at all levels within the design hierarchy. If the *show signals* command is issued before elaboration is completed, then only those signals that have been elaborated so far will be displayed.

*show time*

Displays the current simulation time and cycle count.

*show objects*

Displays the name and class of the objects defined in the context region. Objects include: constants, variables, and signals.

## ❑ Breakpoint Control Commands

*break process { <process_name> | * }*

Inserts a breakpoint at the resumption of the process *<process_name>*. (See the command *show processes* to determine the names of processes within the model.) If an asterisk ("*") is specified instead of a specific process name, then a breakpoint is inserted at the resumption of any process.

*break line { [ <source_file_name> `] <number> | . }*

Inserts a breakpoint just prior to the execution of the declaration or statement at the specified line number. If the optional file is not specified, then a breakpoint is inserted at the specified line number in the current source file. If a period (".") is specified instead of a specific line number, then a breakpoint is inserted at the current line number. (See the *show context* command to display the name of the current source file.)

*break suspend*

Inserts a breakpoint just after the suspension of every process.

*break time <absolute_time>*

Inserts a breakpoint at the specified absolute simulation time. Any valid VHDL time literal may be used to specify the absolute time.

*cancel break { <number> | * }*

Removes a previously inserted breakpoint. The breakpoint to remove is specified by its number in the list of currently active breakpoints. (See the *show breakpoints* command.) If an asterisk ("*") is specified instead of a breakpoint number, then all breakpoints are removed.

*show breakpoints*

Displays a list of the breakpoints that have been established within the simulation.

## ❑ Region Traversal Commands

*show regions*

Displays the list of regions contained by the context region.

*show traceback*

Display the current execution stack. The top of the stack is the top of the design hierarchy. The bottom of the execution stack is the region for the process or subprogram where execution has currently stopped.

*set region { <name> | .. | / | ~ }*

Set the context region to the specified region. The new region is specified relative to the context region. If a simple name is specified, the context region is set to the child region with the simple name. If a double period ("..") is specified, the context region is set to the parent region. If a slash ("/") is specified, the context region is set to the "root region" (the enclosing region of the top-level entity). If a tilde ("~") is specified, the context region is set to the bottom of the current execution stack (the context where execution has currently stopped). (See the show regions command to display the child regions of the context region.)

*show context*

Displays the path name and the VHDL source for the context region. The first line of source from the region is displayed.

*down*

Set the context region to the region down one region in the execution stack.

*up*

Set the context region to the region up one region in the execution stack.

*examine { <simple_name> | * | . }*

Displays the name and current value of all objects with the specified simple name in the "expanded scope" of the context region. The expanded scope includes the context region, all enclosing regions of the context region and packages. If an asterisk ("*") is used, then all the objects in the expanded scope are displayed. If a period (".") is specified, then all the objects in the context region are displayed. (See the command *set region* to set the context region. See the command *show context* to display the path name of the context region.)

## ❑ Signal Trace Control Commands

*set tracing { on | off }*

Controls the recording of all data to the signal trace file. If the value is set to *off* no data is written to the signal trace file. If the value is *on* data is written to the signal trace file. Signal trace file data includes type declarations, signal declarations, assertion messages, error messages, simulation time, signal transactions and signal events. The default setting is *on*.

*trace signals { <signal_number> [ - <signal_number> ] | * } ...*

Specifies a list of signals for which trace data will be recorded. If data is currently being written to the signal trace file, the declaration, transaction and event information will be included for the listed signals. The asterisk ("*") specifies that all the signals in the simulation model will be traced. (See the *set tracing* command to control the recording of all trace data.)

Signals can be added to or deleted from the traced set at anytime during simulation. However, signals not traced during elaboration can not be traced later in simulation.

A contiguous range of signal numbers can be specified by using the range notation: <*signal_number*>-<*signal_number*>. For example, the command "trace signals 1 - 3 14" will trace the signals 1, 2, 3, and 14.

The *show signals* command marks each signal currently being traced with an asterisk ("*");

*untrace signals { <signal_number> [ - <signal_number>] | * } ...*

Specifies a set of signals for which trace data will not be recorded. The asterisk ("*") specifies trace data for all the signals in the simulation model will not be recorded.

Signals can be added to or deleted from the traced set at anytime during simulation. However, signals not traced during elaboration cannot be traced later in simulation.

A contiguous range of signal numbers can be specified by using the range notation: <*signal_number*>-<*signal_number*>. For example, the command "untrace signals 1 - 3 14" will turn tracing off for the signals 1, 2, 3, and 14.

## ❑ Mode Control Commands

*set expert { on | off }*

Sets the expert mode. If expert mode is turned off, the show commands display more descriptive headings. If expert mode is turned on, only basic information is displayed in the headings. The default setting is *off*.

*set flatten_ports { on | off }*

Controls the optimization of hierarchical nets. If flatten_ports is set to on during elaboration time, simple hierarchical nets are flattened into a single signal. Simple hierarchical nets contain only signals and ports with neither resolution functions or type conversion functions. The default setting is *on*.

Setting *flatten_ports* to off has the same effect as the *-noflatten* switch to the *simulate* command.

*set verbose { on | off }*

Controls information messages displayed by the kernel. If verbose is set to on, information messages are displayed at each simulation cycle. If verbose is set to off, no information message is displayed. The default setting is *off*.

*set limit <cycles>*

Sets the maximum number of delta cycles for the current simulation. Simulation will terminate at the specified delta cycle. After termination region traversal commands and information display commands are available, but simulation cannot be resumed. The default setting is no limit.

# General Commands

These commands include commands to capture and execute command files, and a number of miscellaneous commands.

## ❑ Command File Generation and Execution Commands

*record [ <file_name> ]*

Causes each subsequent simulation command entered by the user to be echoed to a command file. If a file name is specified, then that file name is used; otherwise the default file name "vss.script" is used. The named file is opened, and each subsequent command is echoed to the command file until the *end* command is issued.

The commands *record* and *end* are never echoed to the command file.

*# [ <comment_string> ]*

This command has no effect. However, it will be echoed to the current command file, if recording is in effect. The leading "#" causes this line to be treated as a comment when the command file is processed later.

*end*

Causes command recording to terminate and closes the output command file. If recording is not in effect when the *end* command is issued, the *end* command has no effect.

*do [ <file_name> ]*

Causes simulation commands to be read from a command file. If a file name is specified, then that file name is used; otherwise the default file name "vss.script" is used. The named file is opened, and commands are read from the command file and executed until end of file is reached.

If command recording is in effect when the *do* command is issued, then the *do* command has no effect. The *do* command is never echoed to a command file.

## ❑ Miscellaneous Commands

*help*

Displays information about the available commands.

The *help* command generates a list of each command, its parameters, and a brief description of its function.

*version*

Displays the version of the VHDL Simulation System you are using.

*exit* or *quit*

Causes simulation to terminate.

# CHAPTER 2
# VHDL LANGUAGE SUMMARY

This chapter describes how to write VHDL's design units, declarations, and statements.

The following conventions are used in illustrating VHDL syntax. VHDL's reserved words are shown **emboldened**; non-terminals (which you must replace with other text), are shown in a roman typeface. If a non-terminal contains an *italicized* prefix, the non-terminal is identical to the same non-terminal without the prefix; the prefix conveys some additional information. For example, the non-terminal *library*_simple_name means the same as simple_name; however, the name must be that of a library.

Items enclosed in brackets ([ … ]) are optional; items enclosed in braces ({ … }) indicates that the items may be repeated zero or more times. A list of items separated by vertical bars ( | ) indicate that one item must be selected from the list. All other punctuation (e.g., parentheses, commas, and semicolons) are part of the syntax.

Not all of VHDL is described here. We have selected only those design units, declaration, and statements covered in this manual. In any case, the final authority on all matters concerning VHDL is the IEEE 1076 VHDL Language Reference Manual, published by the IEEE.

## VHDL Lexical Elements

The lexical elements of VHDL are VHDL's "words"—those atomic elements of a description composed of printing characters. VHDL's lexical elements comprise four classes: identifiers and reserved words, character literals, string and bit-string literals, and numeric literals. Aggregates, strictly speaking, are not lexical elements (they can span several lines of a VHDL description); however, they are included here as they are basic elements of a VHDL description.

### ❑ Identifiers and Reserved Words

VHDL's identifiers and reserved words have identical structure. They must begin with an alphabetic character. Subsequent characters are either alphabetic or numeric. In addition, underscores can be used in an identifier, but an underscore may not begin nor may it end an identifier. Additionally, multiple underscores may not appear consecutively in an identifier. Identifiers can be of indefinite length, and all characters are significant; however, the case of the alphabetic characters in an identifier is not significant.

Thus, the following identifiers are legal and distinct: "X", "F1234", "VHDL1076", "VHDL_1076", "A_long_identifier_name", "A_longer_identifier_name".

The following identifiers are legal, but represent the same name: "Alert", "ALERT", "alert".

Finally, the following identifiers are illegal: "74LS00" (begins with a non-alphabetic character), "Q_" (ends with an underscore), "A__Z" (contains multiple, consecutive underscores).

Some identifiers are required by VHDL as the structural elements of VHDL design units, declarations, statements and expressions. These *reserved words* are listed below. Reserved words may not be used except as the structural elements of VHDL descriptions.

**abs**
**access**
**after**
**alias**
**all**
**and**
**architecture**
**array**
**assert**
**attribute**

**begin**
**block**
**body**
**buffer**
**bus**

**case**
**component**
**configuration**
**constant**

**disconnect**
**downto**

**else**
**elsif**
**end**
**entity**
**exit**

**file**
**for**
**function**

**generate**
**generic**
**guarded**

**if**
**in**
**inout**
**is**

**label**
**library**
**linkage**
**loop**

**map**
**mod**

**nand**
**new**
**next**
**nor**
**not**
**null**

**of**
**on**
**open**
**or**
**others**
**out**

**package**
**port**
**procedure**
**process**

**range**
**record**
**register**
**rem**
**report**
**return**

**select**
**severity**
**signal**
**subtype**

**then**
**to**
**transport**
**type**

**units**
**until**
**use**

**variable**

**wait**
**when**
**while**
**with**

**xor**

## ❑ Character Literals

A character literal represents a single, printing character. It is represented as the printing character surrounded by two apostrophe characters. For example, 'A' is the character literal "capital-A"; case is significant in character literals. (The literal is usually pronounced "tick-A-tick.") The space character is represented as ' ' (that is, "tic-space-tic"), and the apostrophe character is represented as '''' (that is, three apostrophes in a row, not four).

Character literals are literals of any enumeration type containing character elements. These include the predefined types Bit and Character.

## ❑ String and Bit String Literals

A *string literal* is a sequence of printing characters. It is represented as the sequence of printing characters surrounded by double-quotes ("). Spaces and tabs

may be included, and the case of the alphabetic characters is significant. Double quotes may be included in a string by doubling them. Thus, the string literal """""" represents a string containing a single, double-quote character. Note that a string literal containing one character is different from a character literal.

Strings may represent any one-dimensional array type whose elements include characters. In particular, they may represent values of the predefined types String and Bit_Vector. When they represent bit strings, they are interpreted as binary bit strings.

Bit strings are essential to hardware description languages, so VHDL includes additional literals specifically for representing bit strings in various bases. A *bit string literal* consists first of a base specifier, one of the characters B (denoting a binary bit string literal), O (an octal bit string literal), or X (a hexadecimal bit string literal)—case is immaterial. Then a string comprising only characters in the set {0, 1, _} for binary bit strings, {0, 1, 2, 3, 4, 5, 6, 7, _} for octal bit strings, or {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, a, b, c, d, e, f, _} for hexadecimal bit strings, follows. (Again, case is immaterial in hexadecimal bit strings.) The underscore character may be used to group bits in the string and does not affect the value of the bit string.

Thus, the following bit strings all represent the same value: "001110100000", B"001110100000", b"001_110_100_000", B"0011_1010_0000", O"1640", x"3A0", X"3_A_0".

Bit string literals represent values of any one-dimensional array type whose elements are of the predefined type Bit; notably, the predefined type Bit_Vector. Note that octal bit strings are always a multiple of three bits long and that hexadecimal bit strings are always a multiple of four bits long.

## ❑ Numeric Literals

*Numeric literals* represent integer and real number values. They may be represented in any base between 2 and 16, inclusive.

Base-10 (decimal) numbers, are conventionally represented, as a sequence of digits from 0 through 9, inclusive. Underscores may be used to group the digits; if used, they do not affect the value of the number. If the number is a real number, a decimal point and a fractional part must follow, even if the fractional part is zero; the same is true of the integral part if it is zero. (It is the decimal point that distinguishes integers from real numbers.) Finally an optional exponent may follow: the character E (in either upper- or lower-case) followed by an integer exponent. The exponent must not be negative if the number is an integer.

Some decimal numbers are: 12, 0, 1.0, 0.1, 1e6, 1.076E+3, 123_456, 3.141_593, 10.0E-02.

Literals in other bases may be represented as *based literals*. A based literal begins with a *base specifier*, an integer in the range 2 through 16. Then, a number is specified in digits appropriate to the base, between number signs (#). As is true of decimal numbers, the number may include underscores to group the digits. It may also include a radix point and a fractional part, if it is a real number. Finally, an exponent specification, E followed by an integer exponent, may follow the second number sign. Note that the exponent is relative to the base of the

number; that is, the rest of the number is multiplied by the base of the number to the power of the exponent.

Some based literals are: 2#110_1010#, 16#CA#, 16#f.ff#e+2, 10#1076#.

## ❑ Aggregates

An aggregate represents a value of any array or record type. It consists of a sequence of associations, separated by commas and surrounded by parentheses. If the aggregate's type is a one-dimensional array, then the associations associate the elements of the array; if the aggregate's type is an $n$-dimensional array (with $n > 1$), then the associations are arrays of dimension $n$-1. Finally, if an aggregate is of a record type, the associations associate elements of the record.

Each association has an optional *formal* part, which is the index being associated (in the case of an array aggregate), the name of the element being associated (in the case of a record aggregate) or the reserved word **others**, followed by an association arrow, =>. The *actual* part of the association then follows. Associations without a formal part are called *positional associations*; associations with a formal part are *named associations*. All positional associations must precede any named associations in an aggregate. An *others association* (one with the reserved word **others** appearing in the formal) must be the last association in an aggregate. An others association covers all elements not otherwise associated (either explicitly, with named associations, or implicitly, with positional associations) by the aggregate.

Some examples of aggregates are:

```
(1, 2, 3, 4, 5)
```

A five-element array of integers, whose left-most element is a 1, and whose right-most element is a 5. This aggregate could also be the value of a record, all of whose elements are integers. The first element is a 1, and the last element is a 5.

```
(Day => 21, Month => Dec, Year => 1988)
```

A value of a record type having three fields. Two of them, "Day" and "Year", are integers; the other, "Month" is an enumerated type.

```
('1', '0', '0', '1', others => '0')
```

A vector whose elements are bits. The first four bits are "1001"; the remainder (no matter how many there are) are zero-filled.

```
(('X', '0', 'X'), ('0', '0', '0'), ('X', '0', '1'))
```

A two-dimensional, three-by-three array of characters. This aggregate could also be a one-dimensional array of three elements, each element of which is another one-dimensional array of three characters.

```
("X0X", "000", "X01")
```

This aggregate as the same aggregate as the previous example.

# Design Units

The design units of VHDL are its compilation units, the smallest piece of VHDL which may be separately compiled. Four of VHDL's five design units are described here; only the configuration declaration is not.

## ❑ Context Clause

Every design unit are preceded by a context clause, consisting of zero or more library clauses and use clauses.

A library clause consists of the reserved word **library** followed by a list of comma-separated library names; the list is terminated by a semicolon:

**library** *library*_simple_name { , *library*_simple_name };

A library name is an identifier that denotes a library defined by the Scout VHDL Simulation System. The name appearing within a VHDL description need not include the "dls_" prefix.

Use clauses are presented in the section on Declarations, below.

## ❑ Entity Declaration

VHDL's entity declaration describes the interface of a model to its environment. It also describes tests and characteristics common to any implementation of the model.

```
context_clause
entity identifier is
    [ generic ( interface_constant { ; interface_constant } ) ; ]
    [ port ( interface_signal { ; interface_signal } ) ; ]
    { declaration }
[ begin
    { statement } ]
end [ entity_simple_name ] ;
```

Any declaration except a variable declaration may appear in an entity interface.

The only statements that may appear in an entity interface are assertion statements, passive procedure calls, and passive process statements. A passive procedure call does not directly or indirectly assign values to any signals; a passive process statement is defined similarly.

## ❑ Architecture Body

VHDL's architecture body describes an implementation of an entity interface. An entity may have zero or more architecture bodies associated with it.

```
context_clause
architecture identifier of entity_name is
    { declaration }
begin
    { statement  }
end [ architecture_simple_name ] ;
```

Any declaration except a variable declaration may appear in an architecture body.

Assert statements, component instantiation statements, simple, conditional and selected signal assignment statements, generate statements, procedure calls, and process statements are the only statements that may appear in an architecture body.

### ❑ Package Declaration

A package declaration groups a set of declarations so that they may be manipulated as a whole.

```
context_clause
package identifier is
    { declaration }
end [ package_simple_name ] ;
```

Any declaration except a function body, procedure body, or variable declaration may appear in a package declaration.

A package declaration may not contain any statements.

### ❑ Package Body

A package body is required for any package declaration containing a function declaration or a procedure declaration. The package body must contain a function body for each function declared in either the package interface or body; moreover, the package body must also contain a procedure body for each procedure declared in the package interface or body.

```
context_clause
package body package_simple_name is
    { declaration }
end [ package_simple_name ] ;
```

A package body may contain any declaration except a variable declaration; it may not contain any statements.

## Declarations

VHDL's declarations create named items that can later be used in statements. The use clause, strictly speaking, is not a declaration but is included with the declarations because it can appear anywhere a declaration can appear.

Note that the interface declarations are another special case. Interface declarations may appear only within the ports and generics of entity declarations,

component declarations, and block statements (not described in this chapter), and within the parameter lists of function declarations, function bodies, procedure declarations, and procedure bodies.

## ❑ Component Declaration

A component declaration declares a component that can be later instantiated with a component instantiation statement.

```
component identifier
    [ generic ( interface_constant { ; interface_constant } ) ; ]
    [ port ( interface_signal { ; interface_signal } ) ; ]
end component ;
```

During model building, the Scout VHDL Simulation System binds each component instance with an entity-architecture pair to form a complete simulation model.

## ❑ Constant Declaration

A constant is a named value repository that may be used in expressions. The value of a constant may not be modified after the constant has been created.

```
constant identifier { , identifier } : type_mark [ constraint ] := expression ;
```

A type mark is the name of a type or subtype. If the type mark denotes an unconstrained array type (e.g., Bit_Vector), then a constraint supplying the bounds for each dimension of the array must be supplied, unless the bounds can be derived from the expression.

A constraint may also be specified if the type mark denotes a scalar type; this constraint supplies a further constraint to the type mark. For example, the constraint in the following constant declaration requires the constant to have an integral value in the range 0 to 100, inclusive:

```
constant Passing_Grade: Natural range 0 to 100 := 75;
```

## ❑ Function Body

A function body provides the implementation of a function. If no function interface has previously appeared for this function, it also defines its interface.

```
function designator [ ( interface_declaration { ; interface_declaration } ) ]
    return type_mark
is
    { declaration }
begin
    { statement  }
end [ designator ] ;
```

The designator of a function may be either an identifier or a string. If it is a string, it must be one of the following: “&”, “*”, “**”, “+”, “-”, “/”, “/=”, “<”, “<=”, “=”, “>”, “>=”, “abs”, “and”, “mod”, “nand”, “nor”, “not”, “or”, “rem”, and “xor”; the function then overloads one of VHDL’s predefined operators.

The interface declarations of a function must be either interface constants or interface variables; moreover, they must all be of mode **in**.

If the function has a declaration separate from the body, then the function's specification (the part prior to the reserved word **is**) should be identical to the specification appearing in the function's declaration. This property is called *conformance*.

Any declaration except for a component declaration or a signal declaration may appear in the function body.

Only assert statements, case statements, loops, if statements, null statements, procedure calls, return statements, simple signal assignment statements, and variable assignment statements may appear in a function body.

A function must terminate its execution with a return statement.

## ❑ Function Declaration

A function declaration declares the name, return type, and list of parameters for a function. A function body subsequently must be provided to supply the implementation of the function.

**function** designator [ ( interface_declaration { ; interface_declaration } ) ]
  **return** type_mark ;

See the Function Body section of this chapter for the restrictions on the interface declarations and on the need for conformance between the function declaration and the function's body.

## ❑ Interface Declaration

There are three kinds of interface declarations: interface constant declarations, interface signal declarations, and interface variable declarations.

[ **constant** ] identifier { , identifier } : [ **in** ] type_mark [ constraint ]
  [ := expression ]

Interface constants are used to pass information down the levels of a design hierarchy (these constants are called *generic constants*) or from a caller into a function or procedure (these are *constant parameters*). See the Constant Declaration section of this chapter, above, for information on the type mark and constraint.

[ **signal** ] identifier { , identifier } :
  [ mode ] [ *resolution_function*_name ] type_mark [ constraint ] [ **bus** ]
    [ := expression ]

Interface signals are used to pass information between components and processes within a design hierarchy (ports) or between a caller and a function or procedure (signal parameters). The mode of these signals can be any of **in**, **out**, **inout**, or **buffer**, except that signal parameters to functions must be of mode **in** and signal parameters to procedures cannot be of mode **buffer**.

See the Constant Declaration section of this chapter, above, for information on the type mark and constraint. The use of the resolution function name or reserved word **bus** is not discussed in this manual.

[ **variable** ] identifier { , identifier } : [ mode ] type_mark [ constraint ]
[ := expression ]

Interface variables are used to pass information between a caller and a procedure; such variables are always variable parameters. The mode of these variables is always one of **in**, **out**, or **inout**.

See the Constant Declaration section of this chapter, above, for information on the type mark and constraint.

## ❑ Procedure Body

A procedure body provides the implementation of a procedure. If no procedure interface has previously appeared for this procedure, it also defines its interface.

**procedure** identifier [ ( interface_declaration { ; interface_declaration } ) ] **is**
{ declaration }
**begin**
{ statement }
**end** [ *procedure*_simple_name ] ;

If the procedure has a declaration separate from the body, then the procedure's specification (the part prior to the reserved word **is**) should be identical to the specification appearing in the procedure's declaration. This property is called *conformance*.

Any declaration except for a component declaration or a signal declaration may appear in the procedure body.

Only assert statements, case statements, loops, if statements, null statements, procedure calls, return statements, simple signal assignment statements, variable assignment statements, and wait statements may appear in a procedure body.

A procedure may, but need not, terminate its execution with a return statement; "falling through" to the **end** of the procedure body will terminate its execution.

## ❑ Procedure Declaration

A procedure declaration declares the name and list of parameters for a procedure. A procedure body subsequently must be provided to supply the implementation of the procedure.

**procedure** identifier [ ( interface_declaration { ; interface_declaration } ) ] ;

See the Procedure Body section of this chapter for the restrictions on the interface declarations and on the need for conformance between the procedure declaration and the procedure's body.

## ❑ Signal Declaration

A signal is a named value repository that may be both read from and assigned to. Assignment to a signal does not take effect immediately, but only after some time delay. The past history of signal values may be read (with the predefined attributes of signals), and pending changes of the signal's value may be overridden by subsequent assignments to the signal.

Signals propagate time-variant values between the constituents of a model.

**signal** identifier { , identifier } :
[ *resolution_function*_name ] type_mark [ constraint ] [ **register** | **bus** ]
[ := expression ] ;

See the Constant Declaration section of this chapter, above, for information on the type mark and constraint. The use of the resolution function name or reserved words **register** or **bus** is not discussed in this manual.

## ❑ Subtype Declaration

A subtype declaration declares a subtype of another type or subtype.

**subtype** identifier **is** [ *resolution_function*_name ] type_mark [ constraint ] ;

See the Constant Declaration section of this chapter, above, for information on the type mark and constraint. The use of the resolution function name is not discussed in this manual.

## ❑ Type Declaration

A type declaration declares a type: a set of values and a set of operations on those values.

**type** identifier [ **is** type_definition ] ;

A type definition is either a scalar type definition (integers, reals, enumerations, and physical types are the scalar types), a composite type definition (arrays and records are the composite types), an access (pointer) type definition, or a file type definition. The form of type declaration without a type definition is an *incomplete type declaration*; such declarations are useful in the declaration of access types.

A full discussion of VHDL's type model is beyond the scope of this manual; see the VHDL code in Chapter 3 for some examples of VHDL type declarations.

## ❑ Use Clause

A use clause "imports" names into the current design unit.

**use** use_name { , use_name } ;

A use name takes on one of the following six forms:

*library*_simple_name . *design_unit*_simple_name

This form of use name imports the name of the design unit from the designated library. The library must have been specified with a library clause.

*library*_simple_name . **all**

This form of use name imports the names of all design units in the designated library. The library must have been specified with a library clause. Use this form with caution as you may import many names!

*library*_simple_name . *package*_simple_name . declaration_designator

This form of use name imports the designated declaration from the given package, which must be found in the designated library. The library must have been specified with a library clause.

*library*_simple_name . *package*_simple_name . **all**

This form of use name imports all declarations from the given package, which must be found in the designated library. The library must have been specified with a library clause.

*package*_simple_name . declaration_designator

This form of use name imports the designated declaration from the given package, which must have previously been imported by another use name.

*package*_simple_name . **all**

This form of use name imports all declarations from the given package, which must have previously been imported by another use name.

A declaration designator is one of the following:

| | |
|---|---|
| string | The designator denotes one of VHDL's predefined operators or a function overloading one of the operators. |
| character_literal | The designator denotes a character literal from an enumeration type. |
| simple_name | The designator denotes some other declaration, whose name was specified with an identifier. |

## ❑ Variable Declaration

A variable is a named value repository that may be both read from and assigned to. Assignment to a variable takes effect immediately; only the variable's current value is accessible.

Variables can be used as temporary storage in subprograms; each time the subprogram is called, the variable is reinitialized.

Variables can also be used in process statements; such variables retain their value until explicitly assigned to by a variable assignment statement.

**variable** identifier { , identifier } : type_mark [ constraint ] [ := expression ] ;

See the Constant Declaration section of this chapter, above, for information on the type mark and constraint.

# Statements

VHDL's statements specify the organization, data flow, or behavior of models and define the actions of subprograms.

All statements, with the exception of the wait statement, execute in zero simulated time.

## ❑ Assert Statement

The assert statement allows the specification and checking of conditions within a model. Such conditions may include operating conditions, input relationships (either functional or temporal), and error conditions.

```
assert boolean_expression
    [ report string_expression ]
    [ severity severity_expression ] ;
```

Whenever the assert statement executes, the Boolean expression is evaluated. If the expression is false, then the report string and severity, if any, are reported by the Scout VHDL Simulation System. An assert statement appearing directly within an entity interface, architecture body, block statement, or generate statement executes whenever there is an event on any of the signals named in the Boolean expression.

## ❑ Case Statement

A case statement executes one of a set of statements based on the value of some expression. There must be a set of statements corresponding to every possible value of the expression.

```
case expression is
    case_statement_alternative
    { case_statement_alternative }
end case ;
```

Each case statement alternative has the following form:

```
when choice { | choice } =>
    { statement }
```

The statements are executed whenever the case's expression matches any of the choices in the alternative. A choice is one of the following:

| | |
|---|---|
| expression | The case's expression must be of a discrete type or a one-dimensional array type; the choice is either a string, bit string, or a constant expression of the same type. The statements are executed if the value of the case's expression matches this choice's value. |

| | |
|---|---|
| range | The type of the case's expression must be of a discrete type. The statements are executed if the value of the case's expression matches any value in the range. |
| **others** | This choice must be the only choice in the alternative; moreover, the alternative must be the last alternative in the case statement. The statements in this alternative are executed only if the case's expression matches the choices in no other alternatives. |

The statements that can appear in the case statement alternatives are limited to assert statements, case statements, exit statements, if statements, loops, next statements, null statements, procedure calls, return statements, simple signal assignment statements, variable assignment statements, and wait statements.

## ❑ Component Instantiation Statement

A component instantiation statement creates a copy of a previously declared component and "wires" that component up into a model.

*instantiation*_label : *component*_name
[ **generic map** ( association_element { , association_element } ) ]
[ **port map** ( association_element { , association_element } ) ] ;

A label is an identifier which gives a name to the instance; the label is declared by its appearance in the component instantiation statement.

An association element associates either one of the component declaration's generic constants (if the association element appears in the instance's generic map) or one of the component declaration's ports (if the association element appears in the instance's port map).

As is true of aggregates, the elements in the maps may use positional association, named association, or a mixture of the two, as long as all positionally associated elements precede any using named association.

## ❑ Conditional Signal Assignment Statement

VHDL's conditional signal assignment statement allows one of a series of waveforms to be assigned to a target, depending on the sequential evaluation of a series of Boolean expressions.

[ label : ]
target <= [ **guarded** ] [ **transport** ]
{ waveform **when** *boolean*_expression **else** } waveform ;

The target of a signal assignment statement is either a signal or an aggregate of signals. In the latter case, the value is distributed across the signals. When the conditional signal assignment statement executes, it evaluates the Boolean expressions in the textual order of their appearance, until it finds one that is True; the waveform associated with that Boolean expression is then assigned to the target. If none of the Boolean expressions are True, the terminal waveform is then assigned to the target.

A waveform consists of a sequence of waveform elements, separated by commas:

waveform_element { , waveform_element }

A waveform element consists of a value expression and an optional time expression:

*value*_expression [ **after** *time*_expression ]

The time expression defaults to 0 ns. The time expressions must be non-negative and in strictly increasing order.

Evaluation of the conditional signal assignment statement occurs whenever an event occurs on any signal in any of the Boolean expressions or value expressions. The use of the reserved words **guarded** or **transport** is beyond the scope of this manual.

## ❑ Exit Statement

The exit statement terminates the execution of a loop; it can only appear directly or indirectly within loops.

**exit** [ *loop*_label ] [ **when** *boolean*_expression ] ;

If the loop label is supplied, the exit statement exits the named loop; it is an error if the exit statement is not directly or indirectly enclosed within the named loop. If the loop label is omitted, the exit statement exits the innermost enclosing loop.

If the **when** clause (and its associated Boolean expression) is supplied, the exit statement exits the appropriate loop if and only if the Boolean expression evaluates to True.

## ❑ For Loop

The for loop causes the iterative execution of a sequence of statements for a specified number of times. The next statement may be used to terminate the execution of an iteration; the exit statement may be used to terminate the execution of the entire loop.

[ *loop*_label : ] **for** identifier **in** range **loop**
{ statement }
**end loop** [ *loop*_label ] ;

The loop identifier is a constant that successively takes on values from the range, one value per iteration. The identifier is declared by its appearance in the loop and cannot be accessed from outside the loop, nor can it be assigned to from within the sequence of statements within the loop.

The only statements that may appear within the loop are assert statements, case statements, exit statements, if statements, loops, next statements, null statements, procedure calls, return statements, simple signal assignment statements, variable assignment statements, and wait statements.

## ❑ Generate Statement

The generate statement causes either the repetitive or conditional copying of its enclosed statements.

*generate*_label :
    generation_scheme **generate**
        { statement }
    **end generate** [ *generate*_label ] ;

The generation scheme is either a *for generation scheme*:

**for** identifier **in** discrete_range

which causes the repetitive copying of the generate statements enclosed statements, or an *if generation scheme*:

**if** *boolean*_expression

which causes the conditional generation of the generate statement's enclosed statements.

Akin to the identifier in a for loop, the identifier in a generate statement with a for generate scheme exists only within the generate statement and is a constant within that statement. This constant, which successively takes on values from the supplied range, may be used to supply values to any of the statements enclosed by the generate statement.

The if generation scheme makes a single copy of the enclosed statements if and only if the Boolean expression in the scheme is true.

The statements that may be enclosed within a generate statement are limited to assertion statements, block statements, component instantiation statements, conditional, selected, and simple signal assignment statements, generate statements, procedure calls, and process statements.

## ❑ If Statement

The if statement causes the conditional execution of a sequence of statements.

**if** *boolean*_expression **then**
    { statement }
{ **elsif** *boolean*_expression **then**
    { statement } }
[ **else**
    { statement } ]
**end if** ;

When the if statement executes, the Boolean expression in the if clause is first evaluated. If it is True, the statements within the if clause are evaluated, then the if statement terminates. If the if clause's Boolean expression is False, then, the Boolean expressions in any elsif clauses are successively evaluated until one is found to be True; the statements associated with that clause are then executed. If none of the Boolean expressions are True, then if an else clause is present, the sequence of statements associated with the else clause is evaluated.

The statements appearing within any clause of an if statement can be any of the following: assert statements, case statements, exit statements, if statements, loops, next statements, null statements, procedure calls, return statements, simple signal assignment statements, variable assignment statements, and wait statements.

## ❑ Indefinite Loop

The indefinite loop causes the repetitive, indefinite execution of a sequence of statements.

```
[ loop_label : ] loop
        { statement }
    end loop [ loop_label ] ;
```

The only statements that may appear within the loop are assert statements, case statements, exit statements, if statements, loops, next statements, null statements, procedure calls, return statements, simple signal assignment statements, variable assignment statements, and wait statements.

This loop executes indefinitely unless it executes either an exit statement referring to itself or an outer loop, if it executes a next statement referring to an outer loop, or if it executes a return statement.

## ❑ Null Statement

The null statement does nothing at all.

```
null ;
```

It is often used in the statement part of a case statement alternative in which no action is required. It can also be used as a place holder during code development.

## ❑ Procedure Call

The procedure call invokes the named procedure.

```
[ label : ] procedure_name [ ( association_element { , association_element } )
] ;
```

The label may appear only when the procedure call appears directly within an entity interface, architecture declaration, block statement, or generate statement.

The association elements associate the procedure's interface constants with expressions, its interface variables with variables, and its interface signals with signals; both positional and named association may be used -- see the constraints detailed under Component Instantiation Statement.

## ❑ Process Statement

VHDL's process statement creates a process, which executes in parallel with all other processes and component instances.

[ *process*_label : ]
**process** [ ( *signal*_name { , *signal*_name } ) ]
{ declaration }
**begin**
{ statement }
**end process** [ *process*_label ] ;

The optional, parenthesized list of signals following the reserved word **process** is called the *sensitivity list*. If present, the sensitivity list causes the process to execute once through its list of enclosed statements each time there is an event on any of the signals named in the sensitivity list. Processes with sensitivity lists may not contain wait statements, nor may they call procedures that contain wait statements.

All declarations with the exception of component declarations, and signal declarations may appear within a process statement.

The statements that may appear within the process are limited to: assert statements, case statements, if statements, loops, null statements, procedure calls, simple signal assignment statements, variable assignment statements, and wait statements.

## ❑ Return Statement

The return statement terminates the execution of the enclosed function or procedure. The return statement must appear directly or indirectly within a function or procedure body.

**return** [ expression ] ;

The expression can appear only on a return statement appearing within a function body; the type of the expression must match the return type of the function.

## ❑ Selected Signal Assignment Statement

The selected signal assignment statement selects one of a set of waveforms to assign to a target, based on the value of a selector expression.

**with** expression **select**
target <= [ **guarded** ] [ **transport** ] { waveform **when** choice { | choice } ,
}
waveform **when** choice { | choice } ;

The target of the signal assignment statement is either a signal or an aggregate of signals; in the latter case, the assigned value is distributed across the signals.

The expression works like the expression in the case statement: When the selected signal assignment statement executes it evaluates the expression and finds the choice whose value matches that of the expression; the waveform associated with this choice is assigned to the target. See the Case Statement section of this chapter for a description of the choices.

The use of the reserved words **guarded** or **transport** is beyond the scope of this manual.

## ❑ Simple Signal Assignment Statement

The simple signal assignment statement unconditionally assigns a waveform to a target.

target <= [ **transport** ] waveform ;

Descriptions of the target and waveform can be found in the Concurrent Signal Assignment section of this chapter. The use of the reserved word **transport** is beyond the scope of this manual.

## ❑ Variable Assignment Statement

The variable assignment statement assigns a value to a target.

target := expression ;

The target is either a variable or an aggregate of variables; in the latter case, the value is distributed among the variables comprising the target. The expression's type must match that of the target.

## ❑ Wait Statement

The wait statement synchronizes a process with its environment.

**wait**
[ **on** *signal*_name { , *signal*_name } ]
[ **until** *boolean*_expression ]
[ **for** *time*_expression ] ;

The wait statement has three optional clauses, which may be used in the following combinations with the following meanings:

**wait**;

This wait statement suspends the enclosing process permanently; the process will never resume execution.

**wait on** *signal*_name { , *signal*_name } ;

This wait statement suspends the enclosing process until there is an event on one of the referenced signals.

**wait until** *boolean*_expression ;

This wait statement suspends the enclosing process until there is an event on one of the signals used in the Boolean expression. At that point, the expression is evaluated. If True, the process resumes; if False, the process resuspends until there is another event on one of the signals used in the Boolean expression.

Note that if the Boolean expression does not contain any signals, this form of wait statement is equivalent to **`wait;`** thus, the wait statement **`wait until`** `True;` never awakens!

**wait for** *time*_expression ;

This wait statement suspends the enclosing process until the amount of time specified by the time expression elapses. The time expression must be non-negative.

**wait on** *signal*_name { , *signal*_name } **until** *boolean*_expression ;

This wait statement suspends the enclosing process until there is an event on one of the signals listed between the reserved words **on** and **until**. The Boolean expression is then evaluated. If True, the process resumes; if False, the process resuspends.

**wait on** *signal*_name { , *signal*_name } **for** *time*_expression ;

This wait statement suspends the enclosing process until there is an event on one of the signals listed between the reserved words **on** and **until**, or until the amount of time specified by the time expression elapses, whenever occurs first. The time expression must be non-negative.

**wait until** *boolean*_expression **for** *time*_expression ;

This wait statement suspends the enclosing process until there is an event on one of the signals used in the Boolean expression, or until the amount of time specified by the time expression elapses, whichever occurs first. If there was an event on one of the signals, the expression is evaluated. If True, the process resumes; if False, the process resuspends. Under no circumstances does the process suspend for longer than the value of the time expression, which must be non-negative.

**wait**
**on** *signal*_name { , *signal*_name }
**until** *boolean*_expression
**for** *time*_expression ;

This most general form of wait statement suspends the enclosing process until there is an event on one of the signals listed between the reserved words **on** and **until**, or until the amount of time specified by the time expression elapses, whichever occurs first. If there was an event on one of the signals, the expression is evaluated. If True, the process resumes; if False, the process resuspends. Under no circumstances does the process suspend for longer than the value of the time expression, which must be non-negative.

# CHAPTER 3
# VHDL PREDEFINED ATTRIBUTES

VHDL offers a number of attributes defined by the language that provide information about a type, subtype, array, signal, or block (including an architecture). Each attribute is accessed by an *attribute name*, consisting of a prefix, an apostrophe, and the attribute itself. The prefix must be a type, subtype, array, signal, or block for which the attribute is defined.

The predefined attributes are listed by the prefix to which they apply.

## Attributes of Types and Subtypes

VHDL defines the following attributes of types and subtypes. The prefix of such attributes must be the name of a type or a subtype.

### ❑ Base Type

The following attribute returns the base type of a type or subtype.

#### T'Base

T'BASE returns the *base type* of its prefix. The base type of a given type is the type itself. For example, the base type of the predefined type Integer is Integer. The base type of a subtype, is the type from which the subtype is derived, either directly or indirectly. For example, the base type of the predefined subtype Natural is Integer. The base type of a subtype of Natural is also Integer.

Note that T'Base can only be used as the prefix to another attribute.

### ❑ Scalar Subtype Bounds

The following attributes apply to any scalar type or subtype, and return one of the bounds defining the limits of the type or subtype. A *scalar* type or subtype is any integer, real, enumeration, or physical type or subtype.

In each case, the result is the value of the appropriate bound, expressed as the same type as the prefix type or subtype.

For the purposes of illustrating these attributes, we define the following types and subtypes:

```
type T1 is 1 to 10;
```

```
type T2 is 10 downto 1;

subtype S1 is T1 range 3 to 5;
subtype S2 is T1 range 4 downto 3;

subtype S3 is T2 range 1 downto 2;        -- a null subtype!
```

### T'Left

T'LEFT returns the leftmost, or first, bound of the prefix type or subtype.

Thus, T1'Left is 1, T2'Left is 10, S1'Left is 3, S2'Left is 4, and S3'Left is 1. (Note that S1'Base'Left is 1, as S1'Base is T1.)

### T'Right

T'RIGHT returns the rightmost, or last, bound of the prefix type or subtype.

Thus, T1'Right is 10, T2'Right is 1, S1'Right is 5, S2'Right is 3, and S3'Right is 2.

### T'High

T'HIGH returns the largest bound of the prefix type or subtype.

Thus, T1'High is 10, T2'High is also 10, S1'High is 5, S2'High is 4, and S3'High is 2.

Note that T'High = T'Right for a type or subtype defined with an ascending range, unless the type or subtype is null (i.e., it has no members), in which case T'High = T'Left. Similarly, T'High = T'Left for a non-null type or subtype defined with a descending range.

### T'Low

T'LOW returns the smallest bound of the prefix type or subtype.

Thus, T1'Low is 1, T2'Low is also 1, S1'Low is 3, S2'Low is 3, and S3'Low is 1.

Note that T'Low = T'Left for a type or subtype defined with an ascending range, unless the type or subtype is null (i.e., it has no members), in which case T'Low = T'Right. Similarly, T'Low = T'Right for a non-null type or subtype defined with a descending range.

## ❑ Position–Value Conversion

The following attributes allow conversion between the value of a type or subtype and the value's *position number*, the number of values between T'Left for the type or subtype and the current value.

The prefix of these attributes must be an integer, enumeration, or physical type or subtype.

### T'Pos(X)

T'POS(X) returns X's position number within the type or subtype T. X, the *parameter* to the attribute, must be a member of T. The result is an integer, ranging from zero (T'Pos(T'Left) ≡ 0), to one less than the number of values in T.

Using the example types and subtypes from the previous section, T1'Pos(4) is 3 (because T1'Pos(1) is 0), T2'Pos(2) is 8, and S1'Pos(4) is 1. Note that S3'Pos(1) does not have a value, as S3 is a null subtype.

**T'Val(X)**

T'VAL is the inverse of T'Pos. That is, T'Val returns the value of T whose position number is X. Note that T'Val (T'Pos(X)) ≡ X, for any X for which T'Pos(X) has a value. The parameter to T'Val can be of any integer type.

Again using the example types and subtypes from the previous section, T1'Val(0) is 1, and T2'Val(T1'Pos(S2'Left) is 7 (S2'Left is 4, and T1'Pos(4) is 3; thus T2'Val(3) is 7).

### ❑ Value Navigation

The following four attributes allow values to be incremented and decremented within a type. All attributes take a integer, enumeration or physical type or subtype as a prefix, and a value from within the prefix type or subtype as a parameter. In each case, the returned value, if the value exists, is another value within the base type of the prefix type or subtype.

**T'Succ(X)**

T'SUCC(X) returns the successor of X from within T'Base. Thus, T'Succ(X) ≡ T'Base'Val(T'Pos(X) + 1), provided that X ≠ T'Base'High. Note that an error occurs if an attempt is made to evaluate T'Succ(T'Base'High).

**T'Pred(X)**

T'PRED(X) returns the predecessor of X from within T'Base. Thus, T'Pred(X) ≡ T'Base'Val(T'Pos(X) - 1), provided that X ≠ T'Base'Low. Note that an error occurs if an attempt is made to evaluate T'Pred(T'Base'Low).

**T'LeftOf(X)**

T'LEFTOF(X) is the same as T'Succ(X) for non-null, ascending types and subtypes and for null, descending types and subtypes. Similarly, T'LeftOf(X) is the same as T'Pred(X) for non-null, descending types and subtypes and for null, ascending types and subtypes. Note that, in every case, an error occurs if an attempt is made to evaluate T'LeftOf(T'Base'Left).

**T'RightOf(X)**

T'RIGHTOF(X) is the same as T'Pred(X) for non-null, ascending types and subtypes and for null, descending types and subtypes. Similarly, T'RightOf(X) is the same as T'Succ(X) for non-null, descending types and subtypes and for null, ascending types and subtypes. Note that, in every case, an error occurs if an attempt is made to evaluate T'RightOf(T'Base'Right).

## Attributes of Arrays

VHDL defines the following attributes on arrays. The prefix of such attributes must be an array object (i.e., a signal, variable, or constant of an array type), an array subtype denoting a *constrained array subtype*, an access type or variable denoting one of the foregoing, or an alias denoting one of the foregoing. (A constrained array subtype is a subtype denoting an array type that also specifies constraints on the indices of the array.)

In each case, the attribute takes an optional parameter. This parameter must be between one and the number of dimensions in the array and defaults to one. The parameter is used to select which dimension of the array (starting from the leftmost dimension) for which the attribute is obtaining information.

To better illustrate these attributes, we define the following array types and subtypes:

```
type A1 is array (1 to 10, 10 downto 1) of Integer;
subtype Word is Bit_Vector (31 downto 0);
```

## ❑ Array Index Bounds

The following four attributes allow access to the bounds of the indices of arrays.

### A'Left [(N)]

A'LEFT(N) returns the left bound of the *N*th index range of A.

Thus, A1'Left ( = A1'Left(1)) is 1, A1'Left(2) is 10, and Word'Left is 31. Note that it is an error to attempt to compute Word'Left(2), as Word is a one-dimensional array.

### A'Right [(N)]

A'RIGHT(N) returns the right bound of the *N*th index range of A.

Thus, A1'Right is 10, A1'Right(2) is 1, and Word'Right is 0.

### A'High [(N)]

A'HIGH(N) returns the upper bound of the *N*th index range of A.

Thus, A1'High is 10, A1'High(2) is also 10, and Word'High is 31.

### A'Low [(N)]

A'LOW(N) returns the lower bound of the *N*th index range of A.

Thus, A1'Low is 1, A1'Low(2) is also 1, and Word'Low is 0.

## ❑ Array Index Ranges

The following three attributes allow access to the entire range of values defining the index of a particular dimension of an array. They are typically used within for loops and for generates to iterate across an entire dimension of an array, and within type and subtype specifications to declare other arrays whose indices match in some way the prefix array.

### A'Range [(N)]

A'RANGE(N) returns the range of the *N*th index range of A; the values are presented in the same order as in the *N*th index range.

Thus, A1'Range is 1 to 10, A1'Range(2) is 10 downto 1, and Word'Range is 31 downto 0.

#### A'Reverse_Range [(N)]

A'REVERSE_RANGE(N) returns the range of the *N*th index range of A; the values are presented in the order opposite to that of the *N*th index range.

Thus, A1'Reverse_Range is 10 downto 1, A1'Reverse_Range(2) is 1 to 10, and Word'Reverse_Range is 0 to 31.

#### A'Length [(N)]

A'LENGTH(N) returns the number of elements in the *N*th index range of A. Note that it returns zero for a *null array*, one that has at least one index range that is a null range. (A null range, like a null type or subtype, contains no values.)

Thus, A1'Length is 10, as is A1'Length(2). Word'Length is 32.

## Attributes of Signals

VHDL defines the following attributes of signals. The prefix of such attributes must be a signal. Note that some of these attributes themselves return a signal; such a signal is valid as the prefix of another attribute whose prefix must be a signal.

### ❑ Signal-Valued Attributes

The following four attributes return signals. The first two are sensitive to events on the prefix signal, while the final two are sensitive to transactions on the prefix signal.

The first three take an optional parameter of type time, which must not be negative. This parameter defaults to 0 ns.

Signal-valued attributes may not be used inside of functions or procedures.

#### S'Delayed [(T)]

S'DELAYED(T) is a signal of the same type as the base type of S, whose values with respect to time are delayed by T time units. This relationship can be expressed with the following process statement:

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

Assuming that the type of R is the same as the base type of S and that the initial value of R is the same as the initial value of S, then for all T ≥ 0 ns, S'Delayed(T) = R at any time.

Note that S'Delayed(0 ns) ≠ S exactly when S has just changed; i.e., exactly when S'Event is True.

### S'Stable [(T)]

S'STABLE(T) is a Boolean-valued signal that is True precisely when S has not changed value in the last T units of time. S'Stable is False exactly when S'Event is True.

Typical uses for S'Stable are for setup, hold, and other timing checks. For example, the following process checks that the signal Data has been stable for at least $T_{SETUP}$ units of time when the rising edge of Clk occurs:

```
Setup_Check:
   process
   begin
      wait until Clk'Event and Clk = '1';
      assert Data'Stable(TSETUP)
         report "Setup violation";
   end process Setup_Check;
```

The following process checks that the signal Data has been stable for at least $T_{HOLD}$ units of time since the last rising edge of Clk:

```
Hold_Check:
   process
   begin
      wait until Clk'Delayed(THOLD)'Event and Clk'Delayed(THOLD) = '1';
      assert Data'Stable(THOLD)
         report "Hold violation";
   end process Hold_Check;
```

The trick here is that, since we cannot look forward into time from a rising edge of Clk to see if Data will be stable for at least $T_{HOLD}$ units of time, we instead delay Clk by $T_{HOLD}$ units of time. By looking backwards from rising edges of this delayed signal, we can see if Data changed during the hold period.

Finally, we can combine these two checks into a single process:

```
Setup_and_Hold_Check:
   process
   begin
      wait until Clk'Delayed(THOLD)'Event and Clk'Delayed(THOLD) = '1';
      assert Data'Stable(TSETUP + THOLD)
         report "Setup or Hold violation";
   end process Setup_and_Hold_Check;
```

### S'Quiet [(T)]

S'QUIET, like S'Stable, is a Boolean-valued signal. Where S'Stable is sensitive to events (changes) on S, S'Quiet is sensitive to transactions on S. Any assignment to S causes a transaction on S. S'Quiet(T) is True exactly when there has not been an assignment to S during the last T units of time.

Transaction-sensitive attributes are typically used for very-high level modeling.

### S'Transaction

S'TRANSACTION is a Bit-valued signal that toggles whenever there is a transaction on S. Thus S'Transaction turns a transaction into an event.

S'Transaction can be used to sensitize processes to transactions on S, since S'Transaction is itself a signal.

### ❑ Function-Valued Attributes

Since the signal-valued attributes cannot be used within functions and procedures, VHDL provides function-valued attributes with functionality equivalent to the signal-valued attributes. These function-valued attributes can be used inside of functions and procedures.

#### S'Event

S'EVENT is True exactly when S'Stable(0 ns) is False; that is, exactly when S has just changed value.

#### S'Active

S'ACTIVE is True exactly when S'Quiet(0 ns) is False; that is, exactly when there has just been a transaction on S.

#### S'Last_Event

S'LAST_EVENT returns the amount of time that has elapsed since the last change of value on S.

#### S'Last_Active

S'LAST_ACTIVE returns the amount of time that has elapsed since the last transaction on S.

#### S'Last_Value

S'LAST_VALUE returns the value that S had just prior to its last event. (Note that S'Last_Value = S until S has changed value for the first time.)

## Attributes of Blocks

VHDL defines the following attributes of blocks. The prefix of such attributes must be either the name of an architecture or the label of a block statement.

### ❑ Behavioral Description Query

VHDL provides a predefined attribute to detect the occurrence of component instantiation statements within an architecture or a block statement.

#### B'Behavior

B'BEHAVIOR returns True if and only if B *does not* contain a component instantiation statement; i.e., B contains only behavioral statements.

## ❑ Structural Description Query

### B'Structure

B'STRUCTURE returns True if and only if B *does not* contain any signal assignment statements, either within or outside of processes. B'Structure is True exactly when B is a pure structure.

Note that it is possible for both B'Behavior and B'Structure to be False for a given B. However, it is never possible that both B'Behavior and B'Structure are True for a given B.

# CHAPTER 4
# VHDL STANDARD PACKAGES

VHDL provides two standard packages as part of its definition. In addition, other IEEE standardization bodies (see Chapter 4), are developing standard packages outside of VHDL's definition. Some of these packages are explained in this chapter.

## Package Standard

Package Standard (in library Std) is VHDL's predefined environment. The definitions in package standard are provided at the beginning of every compilation, exactly as if the use clause `use Std.Standard.all;` appeared at the start of every design unit.

As defined in the VHDL LRM, package Standard allows implementation-defined bounds for types Integer, Real, and Time. In the description of package Standard below, we have supplied the VSS-defined bounds for these types.

Note that it is never required that package Standard be compiled; indeed, it is impossible to do so.

```
package STANDARD is

    -- predefined enumeration types:

    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);
```

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

    -- predefined numeric types:

    type INTEGER is range -2147483648 to 2147483647;

    type REAL is range -1.7014110e+38 to 1.7014110e+38;

    -- predefined type TIME:

    type TIME is range -9223372036854775808 to 9223372036854775807
        units
            fs;                             -- femtosecond
            ps    =   1000 fs;              -- picosecond
            ns    =   1000 ps;              -- nanosecond
            us    =   1000 ns;              -- microsecond
            ms    =   1000 us;              -- millisecond
            sec   =   1000 ms;              -- second
            min   =   60 sec;               -- minute
            hr    =   60 min;               -- hour
        end units;

    -- function that returns the current simulation time:

    function NOW return TIME;

    -- predefined numeric subtypes:

    subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;

    subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

    -- predefined array types:

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

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

end STANDARD;
```

# Package TextIO

Package TextIO, also in library Std, is the other LRM-defined standard package. It provides a set of input and output facilities between VHDL's standard types and text files. Unlike package Standard, however, no implicit use clause provides direct access to package TextIO's features; the user may supply the use clause `use Std.TextIO.all;` to provide such direct access. Alternatively, the names may be referenced without resorting to a use clause; e.g., the name "Std.TextIO.Line" references the Line definition within TextIO.

It is never necessary to compile package TextIO, although it is possible to do so as TextIO's interface is specified as a normal VHDL package. Note that compiling the package interface will invalidate TextIO's body, which is typically not written in VHDL and therefore cannot be compiled.

The version of TextIO presented in this chapter differs from that published in the LRM. In this version, several bugs, noted by the VASG, have been fixed according to the VASG-approved suggested repair. Specifically, the procedure WriteLine has been repaired by specifying that its first parameter is of class variable and that its second parameter is of mode inout. Moreover, the EndLine function has been deleted as it is specified with illegal VHDL; the expression "L'Length = 0" provides identical functionality.

```
        package TEXTIO is

        -- Type Definitions for Text I/O:

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

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

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

          subtype WIDTH is NATURAL;      -- for specifying widths of output
fields

        -- Standard Text Files:

          file INPUT:  TEXT is in  "STD_INPUT";

          file OUTPUT: TEXT is out "STD_OUTPUT";

        -- Input Routines for Standard Types:

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

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

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

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

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

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

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

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

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


        -- Output Routines for Standard Types:

          procedure WRITELINE (variable F: out TEXT;  L: inout LINE);

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

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

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

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

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

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

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

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

  -- File Position Predicates:

--  function ENDLINE (L: in LINE) return BOOLEAN;

--  function ENDFILE (F: in TEXT) return BOOLEAN;

  end TEXTIO;
```

TextIO is used to do line-by-line textual input-output as follows. The type Text is a text file, consisting of a number of lines. The type Line represents one line from a text file.

To read items from a file, a line is first read from the file using ReadLine. Then the various Read procedures can be used to extract data from the line by the type of data. Each form of Read procedure has an optional "Good" parameter that informs the caller whether the call was able to extract a datum of the specified type from the line; if the form of Read without the "Good" parameter is used and a datum of the specified type is not present next on the line, an error results.

To write items to a file, the items are first written to a line using the various Write procedures. The "Justified" parameter allows control over wither the value is left- or right-justified within the field, which is a minimum of "Field" characters wide. (The field is always wide enough to accommodate the value.) Several of the Write procedures have additional parameters: The Write procedure for Reals also includes the "Digits" parameter, which specifies the minimum number of digits to the right of the decimal point to emit. The Write procedure for Time values includes the "Unit" parameter, which specifies in which unit to write the output values.

Once a line has been composed with the Write procedures, it is deposited in an output file with the WriteLine procedure, which clears the line as a side-effect of its execution.

## Package std_logic_1164

The IEEE Design Automation Subcommittee VHDL Analysis and Standardization Group has approved a standard logic package for ASIC and board-level models. The basic logic types of this package are std_ulogic (a scalar), and std_ulogic_vector (a vector), which implement a nine-state logic model.

These are unresolved types, which means that signals of these types cannot be multiply driven. Signals of the resolved logic types, std_logic and std_logic_array, may be multiply driven, the results are supplied by the function "resolved." All the logic types are provided with the standard VHDL logic operators **and**, **nand**, **or**, **nor**, **xor**, and **not**. In addition, various conversion and edge detection functions are provided.

```
-- ------------------------------------------------------------------
--
--   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" ballotting 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 convertors
    -------------------------------------------------------------------

    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;


-- -------------------------------------------------------------------
--
--   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 | 01/02/91  |
-- ---------------------------------------------------------------------

PACKAGE BODY std_logic_1164 IS
    -----------------------------------------------------------------
    -- local types
    -----------------------------------------------------------------
    TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;
    TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

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

    -----------------------------------------------------------------
    -- tables for logical operations
    -----------------------------------------------------------------

    -- truth table for "and" function
    CONSTANT and_table : stdlogic_table := (
    --      ---------------------------------------------------
    --      |  U    X    0    1    Z    W    L    H    -     |   |
    --      ---------------------------------------------------
            ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
            ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
            ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
            ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
            ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
            ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
```

```
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);

-- truth table for "or" function
CONSTANT or_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -      |   |
--      ---------------------------------------------------
        ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ),  -- | U |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | X |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | 1 |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | Z |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | H |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' )   -- | - |
);


-- truth table for "xor" function
CONSTANT xor_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', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | 1 |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )   -- | - |
);

-- truth table for "not" function
CONSTANT not_table: stdlogic_1d :=
--  ---------------------------------------------
--  |   U    X    0    1    Z    W    L    H    -   |
--  ---------------------------------------------
     ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );

-------------------------------------------------------------------
-- overloaded logical operators ( with optimizing hints )
-------------------------------------------------------------------

FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN (and_table(l, r));
END "and";

FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN  (not_table ( and_table(l, r)));
END "nand";

FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN (or_table(l, r));
END "or";

FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN  (not_table ( or_table( l, r )));
END "nor";

FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
```

```
        RETURN (xor_table(l, r));
    END "xor";

--  function "xnor"  ( l : std_ulogic; r : std_ulogic ) return ux01 is
--  begin
--      return not_table(xor_table(l, r));
--  end "xnor";

    FUNCTION "not"  ( l : std_ulogic ) RETURN UX01 IS
    BEGIN
        RETURN (not_table(l));
    END "not";

    ------------------------------------------------------------------
    -- and
    ------------------------------------------------------------------
    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;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'and' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := and_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "and";
    ------------------------------------------------------------------
    FUNCTION "and"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'and' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := and_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "and";
    ------------------------------------------------------------------
    -- nand
    ------------------------------------------------------------------
    FUNCTION "nand"  ( 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;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nand' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := not_table(and_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nand";
```

```
    -------------------------------------------------------------------
    FUNCTION "nand"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nand' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := not_table(and_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nand";
    -------------------------------------------------------------------
    -- or
    -------------------------------------------------------------------
    FUNCTION "or"  ( 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;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'or' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := or_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "or";
    -------------------------------------------------------------------
    FUNCTION "or"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'or' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := or_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "or";
    -------------------------------------------------------------------
    -- nor
    -------------------------------------------------------------------
    FUNCTION "nor"  ( 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;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nor' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
```

```
                result(i) := not_table(or_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nor";
    -------------------------------------------------------------------
    FUNCTION "nor"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nor' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := not_table(or_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nor";
    -------------------------------------------------------------------
    -- xor
    -------------------------------------------------------------------
    FUNCTION "xor"  ( 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;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'xor' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := xor_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "xor";
    -------------------------------------------------------------------
    FUNCTION "xor"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'xor' operator are not of the same
length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := xor_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "xor";
--  -------------------------------------------------------------------
--  -- xnor
--  -------------------------------------------------------------------
--  ---------------------------------------------------------------------------
--  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 is
--      alias lv : std_logic_vector ( 1 to l'length ) is l;
--      alias rv : std_logic_vector ( 1 to r'length ) is r;
--      variable result : std_logic_vector ( 1 to l'length );
--  begin
--      if ( l'length /= r'length ) then
--          assert false
--          report "arguments of overloaded 'xnor' operator are not of the same
length"
--          severity failure;
--      else
--          for i in result'range loop
--              result(i) := not_table(xor_table (lv(i), rv(i)));
--          end loop;
--      end if;
--      return result;
--  end "xnor";
--  ------------------------------------------------------------------
--  function "xnor"  ( l,r : std_ulogic_vector ) return std_ulogic_vector is
--      alias lv : std_ulogic_vector ( 1 to l'length ) is l;
--      alias rv : std_ulogic_vector ( 1 to r'length ) is r;
--      variable result : std_ulogic_vector ( 1 to l'length );
--  begin
--      if ( l'length /= r'length ) then
--          assert false
--          report "arguments of overloaded 'xnor' operator are not of the same
length"
--          severity failure;
--      else
--          for i in result'range loop
--              result(i) := not_table(xor_table (lv(i), rv(i)));
--          end loop;
--      end if;
--      return result;
--  end "xnor";
    ------------------------------------------------------------------
    -- not
    ------------------------------------------------------------------
    FUNCTION "not"  ( l : std_logic_vector ) RETURN std_logic_vector IS
        ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH ) := (OTHERS => 'X');
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := not_table( lv(i) );
        END LOOP;
        RETURN result;
    END;
    ------------------------------------------------------------------
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH ) := (OTHERS => 'X');
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := not_table( lv(i) );
        END LOOP;
        RETURN result;
    END;
    ------------------------------------------------------------------
    -- conversion tables
    ------------------------------------------------------------------
    TYPE logic_x01_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF X01;
    TYPE logic_x01z_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF X01Z;
    TYPE logic_ux01_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF UX01;
    ------------------------------------------------------
    -- table name : cvt_to_x01
    --
    -- parameters :
    --        in  :  std_ulogic  -- some logic value
    -- returns    :  x01         -- state value of logic value
```

```
    -- purpose    :  to convert state-strength to state only
    --
    -- example    : if (cvt_to_x01 (input_signal) = '1' ) then ...
    --
    -------------------------------------------------------------
    CONSTANT cvt_to_x01 : logic_x01_table := (
                          'X',  -- 'U'
                          'X',  -- 'X'
                          '0',  -- '0'
                          '1',  -- '1'
                          'X',  -- 'Z'
                          'X',  -- 'W'
                          '0',  -- 'L'
                          '1',  -- 'H'
                          'X'   -- '-'
                         );

    -------------------------------------------------------------
    -- table name : cvt_to_x01z
    --
    -- parameters :
    --        in  :  std_ulogic  -- some logic value
    -- returns    :  x01z        -- state value of logic value
    -- purpose    :  to convert state-strength to state only
    --
    -- example    : if (cvt_to_x01z (input_signal) = '1' ) then ...
    --
    -------------------------------------------------------------
    CONSTANT cvt_to_x01z : logic_x01z_table := (
                          'X',  -- 'U'
                          'X',  -- 'X'
                          '0',  -- '0'
                          '1',  -- '1'
                          'Z',  -- 'Z'
                          'X',  -- 'W'
                          '0',  -- 'L'
                          '1',  -- 'H'
                          'X'   -- '-'
                         );

    -------------------------------------------------------------
    -- table name : cvt_to_ux01
    --
    -- parameters :
    --        in  :  std_ulogic  -- some logic value
    -- returns    :  ux01        -- state value of logic value
    -- purpose    :  to convert state-strength to state only
    --
    -- example    : if (cvt_to_ux01 (input_signal) = '1' ) then ...
    --
    -------------------------------------------------------------
    CONSTANT cvt_to_ux01 : logic_ux01_table := (
                          'U',  -- 'U'
                          'X',  -- 'X'
                          '0',  -- '0'
                          '1',  -- '1'
                          'X',  -- 'Z'
                          'X',  -- 'W'
                          '0',  -- 'L'
                          '1',  -- 'H'
                          'X'   -- '-'
                         );

    ---------------------------------------------------------------------
    -- conversion functions
    ---------------------------------------------------------------------
    FUNCTION To_bit       ( s : std_ulogic;        xmap : BIT := '0') RETURN BIT
IS
    BEGIN
            CASE s IS
                WHEN '0' | 'L' => RETURN ('0');
```

```
                WHEN '1' | 'H' => RETURN ('1');
                WHEN OTHERS => RETURN xmap;
            END CASE;
    END;
    -------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0') RETURN
BIT_VECTOR IS
        ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0') RETURN
BIT_VECTOR IS
        ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_StdULogic       ( b : BIT               ) RETURN std_ulogic IS
    BEGIN
        CASE b IS
            WHEN '0' => RETURN '0';
            WHEN '1' => RETURN '1';
        END CASE;
    END;
    -------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        ) RETURN std_logic_vector
IS
        ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) IS b;
        VARIABLE result : std_logic_vector ( b'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN std_logic_vector
IS
        ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : std_logic_vector ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR        ) RETURN std_ulogic_vector IS
        ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) IS b;
        VARIABLE result : std_ulogic_vector ( b'LENGTH-1 DOWNTO 0 );
```

```
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_StdULogicVector ( s : std_logic_vector ) RETURN std_ulogic_vector
IS
        ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    END;

    -------------------------------------------------------------------
    -- strength strippers and type convertors
    -------------------------------------------------------------------
    -- to_x01
    -------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector ) RETURN  std_logic_vector IS
        ALIAS sv : std_logic_vector ( 1 TO s'LENGTH ) IS s;
        VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_x01 (sv(i));
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector IS
        ALIAS sv : std_ulogic_vector ( 1 TO s'LENGTH ) IS s;
        VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_x01 (sv(i));
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_ulogic ) RETURN  X01 IS
    BEGIN
        RETURN (cvt_to_x01(s));
    END;
    -------------------------------------------------------------------
    FUNCTION To_X01  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
        ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
        VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_X01  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
        ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
        VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
```

```
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_X01  ( b : BIT ) RETURN  X01 IS
BEGIN
        CASE b IS
            WHEN '0' => RETURN('0');
            WHEN '1' => RETURN('1');
        END CASE;
END;
-------------------------------------------------------------------
-- to_x01z
-------------------------------------------------------------------
FUNCTION To_X01Z  ( s : std_logic_vector ) RETURN  std_logic_vector IS
    ALIAS sv : std_logic_vector ( 1 TO s'LENGTH ) IS s;
    VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_x01z (sv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_X01Z  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector IS
    ALIAS sv : std_ulogic_vector ( 1 TO s'LENGTH ) IS s;
    VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_x01z (sv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_X01Z  ( s : std_ulogic ) RETURN  X01Z IS
BEGIN
    RETURN (cvt_to_x01z(s));
END;
-------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT ) RETURN  X01Z IS
BEGIN
        CASE b IS
            WHEN '0' => RETURN('0');
```

```
            WHEN '1' => RETURN('1');
        END CASE;
END;
-------------------------------------------------------------------
-- to_ux01
-------------------------------------------------------------------
FUNCTION To_UX01  ( s : std_logic_vector ) RETURN  std_logic_vector IS
    ALIAS sv : std_logic_vector ( 1 TO s'LENGTH ) IS s;
    VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_ux01 (sv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector IS
    ALIAS sv : std_ulogic_vector ( 1 TO s'LENGTH ) IS s;
    VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_ux01 (sv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_UX01  ( s : std_ulogic ) RETURN  UX01 IS
BEGIN
    RETURN (cvt_to_ux01(s));
END;
-------------------------------------------------------------------
FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------------------
FUNCTION To_UX01  ( b : BIT ) RETURN  UX01 IS
BEGIN
        CASE b IS
            WHEN '0' => RETURN('0');
            WHEN '1' => RETURN('1');
        END CASE;
END;

-------------------------------------------------------------------
-- edge detection
-------------------------------------------------------------------
FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
BEGIN
    RETURN (s'EVENT AND (To_X01(s) = '1') AND
```

```
                        (To_X01(s'LAST_VALUE) = '0'));
    END;

    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
    BEGIN
        RETURN (s'EVENT AND (To_X01(s) = '0') AND
                        (To_X01(s'LAST_VALUE) = '1'));
    END;

    -------------------------------------------------------------------
    -- object contains an unknown
    -------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;
    -------------------------------------------------------------------
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;
    -------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN IS
    BEGIN
        CASE s IS
            WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
            WHEN OTHERS => NULL;
        END CASE;
        RETURN FALSE;
    END;

END std_logic_1164;
```

# INDEX


Actual
  in Association List
    Definition, 85

Adder (see Examples, Adder)

After
  in Signal Assignment Statement (see Signal Assignment Statement, After Clause)

Aggregate
  Definition, 54
  Examples, 52
  Named Association
    Examples, 52
  Others
    Definition, 54
    Examples, 52

Architecture Body
  Definition, 21
  Examples, 20, 22, 28, 30, 36, 38, 44, 46, 52, 56, 64, 66, 78, 84, 86
  of test bench, 23

Array
  Concatenation (see Concatenation)
  Indexing
    Examples, 38, 52, 66
  Slice
    as target of assignment, 55
  Slicing
    Examples, 38, 52, 66

Array Type Declaration (see Type Declaration, Constrained Array)

Assert Statement (see also Assertion Violation)
  Definition, 23, 39, 53
  Examples, 22, 38, 52
  Report Clause
    Definition, 39
    Examples, 38, 52
  Severity Clause
    Default
      Definition, 53
    Definition, 53
    Examples, 52

Assertion Violation (see also Assert Statement)
  Definition, 53
  VSS actions in response to, 53

Assignment to signal (see Signal Assignment, Transaction, and Event)

Assignment to variable (see Variable Assignment Statement)

Association
  List
    Named Association
      Definition, 85

Association List
  Actual
    Definition, 85
  Definition, 85
  Examples, 22, 30, 36, 38, 44, 46, 56, 66, 84, 86
  Formal
    Definition, 85
  mixing Named and Positional, 85
  Named Association
    Examples, 84
  of Component Instance, 23
  Positional Association
    Definition, 85

Attribute (see Predefined Attribute)

Behavioral Description
  Definition, 53

Binding
  Default Rules
    Definition, 37
  in Elaboration, 37

Bit
  as enumeration type, 65
  predefined type, 21

Bit_Vector
  predefined type, 37

Buffer
  Mode
    Examples, 84, 86

Case Statement
  Definition, 55
  Examples, 52, 64
  When Clause
    Definition, 55
    Examples, 52, 64

Clock
  Concurrent Procedure

Implementation, 77
Gated
Implementation, 67

Component Declaration
Examples, 22, 30, 36, 38, 44, 46, 56, 66, 72

Component Instance (see Component Instantiation Statement)

Component Instantiation Statement
Definition, 23
Examples, 22, 30, 36, 38, 44, 46, 56, 66, 84, 86
use in a test bench, 23

Components Package (see Examples, Components Package)

Concatenation
Definition, 39
Examples, 38, 52, 66

Concurrency
in architecture bodies, 21

Concurrent Procedure
Body of, 77
Call
Definition, 79
Examples, 78, 86
Definition, 75

Conditional Signal Assignment Statement (see Signal Assignment Statement, Conditional)

Constant Declaration
Examples, 22

Constraint
Definition, 53

Controller (see Examples, Controller)

Conversion
Functions, 75

D Flip-Flop (see Examples, D Flip-Flop)

Dataflow Description
Definition, 29

Default
severity in Assert Statement, 53

Delay Models
in Signal Assignment
Delta Delay, 29
Inertial Delay, 29
Propagation Delay, 29

Delta Delay
in Signal Assignment, 29
in two-dimensional time model, 29
in Waveform Displays, 29

Description Styles
Behavioral, 53
Dataflow, 29
Equational, 21
Finite-State Machine, 65
Structural, 37

Edge Detection, 54

Elaboration
Default Binding Rules
Definition, 37
Definition, 37

Else
in If Statement (see If Statement, Else Clause)

Elsif
in If Statement (see If Statement, Elsif Clause)

Entity Interface
Definition, 21
Examples, 20, 22, 28, 30, 36, 38, 44, 46, 52, 56, 64, 66, 78, 84, 86
of test bench, 23
Statement Part
Definition, 53
Examples, 52

Enumeration Type
Bit (predefined type) (see also Bit)
Definition, 65
Definition, 65

Enumeration Type Declaration (see Type Declaration, Enumeration), (see Type Declaration, Enumeration)

Equational Description
Definition, 21

Error
in Assert Statement (see Assert Statement, Severity Clause)

Event
Definition, 25
predefined attribute (see Predefined Attribute, Event)

Examples
Adder
Schematic, 42
Simulation Results, 48
Test Bench, 46
VHDL Model, 44
Component Instantiation Statement
Examples, 86
Components Package, 72
Controller
Simulation Results, 68
State Diagram, 62
State Table, 62
Test Bench, 66
VHDL Model, 64
D Flip-Flop
Schematic, 26
Simulation Results, 32
Test Bench, 30
VHDL Model, 28

Full Adder
Simulation Results, 24
Test Bench, 22
Truth Table, 18
VHDL Model, 20
Latch
Schematic, 34
Simulation Results, 40
Test Bench, 38
VHDL Model, 36
Multiplier
Block Diagram, 82
Hierarchy Diagram, 70
Simulation Results, 88
Test Bench, 86
VHDL Model, 84
Shift Register
Simulation Results, 58
Test Bench, 56
Truth Table, 50
VHDL Model, 52
Utility Package
Simulation Results, 80
Test Bench, 78
VHDL Package, 74

Exit Statement
Definition, 67
Examples, 66
When Clause
Definition, 67
Examples, 66

Failure
in Assert Statement (see Assert Statement, Severity Clause)

Finite-State Machine Description
Definition, 65

For
in Generate Statement (see Generate Statement, For Generate)
in Loop Statement (see Loop, For)
in Wait Statement (see Wait Statement, Timeout Clause)

Formal
in Association List
Definition, 85

Full Adder (see Examples, Full Adder)

Function
Body
Definition, 75
Examples, 74
location when Declaration is in a Package
Definition, 75
repetition of Declaration, 75
use of a Return Statement, 77
Call
Examples, 78, 86
Overloaded
Error Conditions, 79
Examples, 78
Resolution, 79
converting types, 75
Declaration
Definition, 75
Examples, 74
repetition in Body, 75
Number of Parameters
overloading characteristic, 75
Overloading
Definition, 75
Error Conditions in Call, 79
Resolution of Call, 79
Parameter
Definition, 75
Parameter Types
overloading characteristic, 75
Parameters
modifying inside of Body, 77
Return Type
overloading characteristic, 75

Gated Clock
Implementation, 67

Generate Statement
Definition, 45
Examples, 44
For Generate
Definition, 45
Examples, 44
If Generate
Definition, 45
Examples, 44

Hierarchical Name
Definition, 25

If
in Generate Statement (see Generate Statement, If Generate)
in If Statement (see If Statement)

If Statement
Definition, 47
Else Clause
Definition, 47
Examples, 46
Elsif Clause
Definition, 47, 54
Examples, 52, 64
Examples, 46, 52, 64, 66, 74
Nesting, 54

Importing Declarations
with a Use Clause, 79

Indefinite Loop (see Loop, Indefinite)

Index Name
as target of assignment, 74
Definition, 39

Inertial Delay
in Signal Assignment, 29

Infinite Loop (see Loop, Infinite)

Integer
Natural (predefined subtype of), 54
predefined type, 54

Interface List
Examples, 20, 22, 28, 30, 36, 38, 44, 46, 52, 56, 66, 72, 74, 86
Mode
Buffer
Examples, 84, 86
Definition, 21
Ports
feedback in mode buffer, 85
feedback in mode out, 29, 85
of Entity Interface, 21
of Mode Buffer
Examples, 84, 86
Unconstrained
Examples, 52, 56, 64, 84
Subprogram Parameters
Definition, 75
Examples, 74
modifying inside of Body, 77
Unconstrained
Definition, 53

Interface Signal (see Interface Declaration, Ports)

Latch (see Examples, Latch)

Left
predefined attribute (see Predefined Attribute, Left)

Length
predefined attribute (see Predefined Attribute, Length)

Loop
For
declaration of iterator, 23
Examples, 22, 38, 46, 56, 74, 78, 86
Indefinite
Definition, 67
Examples, 66, 74
exiting from, 67
use in Concurrent Procedure Body, 77
Infinite (see Infinite Loop)

Matching Element Rule
Definition, 57

Mode
Buffer
Examples, 84
participation in feedback loops, 85
Definition, 21
Out
participation in feedback loops, 29, 85

Multiple Waveform Elements
Definition, 31

Multiplier (see Examples, Multiplier)

Named Association
in Aggregates (see Aggregates, Named Association)
in Association List
Definition, 85

Natural
predefined subtype, 54

Nested Ifs, 54

Note
in Assert Statement (see Assert Statement, Severity Clause)

Others
in Aggregate (see Aggregate, Others)

Overloading
of Subprograms
Definition, 75
Error Conditions in Call, 79
Resolution of Call, 79

Package
comprising a Declaration and a Body, 75
Motivation, 71
when a Body is required, 75

Package Body
Definition, 75
Examples, 74
when required, 75

Package Declaration
Definition, 73, 75
Examples, 72, 74

Parameter
of Subprogram
Definition, 75

Passive Process (see Process Statement, Passive)

Port List (see Interface List)

Port Map (see Association List)

Position Number
Definition, 77

Positional Association
in Association List
Definition, 85

Predefined Attribute
Event
Definition, 54
Examples, 52, 64, 66, 86
Left
Definition, 55
Examples, 52
Length
Definition, 53
Examples, 52
of arrays
Left
Definition, 55

Examples, 52
Length
Definition, 53
Examples, 52
Range
Examples, 22
Right
Definition, 77
Examples, 74
of scalar types
Val
Definition, 77
Examples, 74
of signals
Event
Definition, 54
Examples, 52, 64, 66, 86
prefix of
Definition, 53
Range
Examples, 22
Right
Definition, 77
Examples, 74
Val
Definition, 77
Examples, 74

Prefix
of Predefined Attribute
Definition, 53

Procedure
Body
Definition, 75
Examples, 74
location when Declaration is in a Package
Definition, 75
repetition of Declaration, 75
use of a Return Statement, 77
Call
Concurrent
Examples, 78, 86
Concurrent
Body of, 77
Definition, 75
Declaration
Definition, 75
Examples, 74
repetition in Body, 75
Number of Parameters
overloading characteristic, 75
Parameter
Definition, 75
Parameter Types
overloading characteristic, 75
Return Type
overloading characteristic, 75

Process Statement
Definition, 23
Examples, 22, 30, 38, 46, 52, 56, 64, 66, 78, 86
Passive
Definition, 53
Sensitivity
in Until Clause of Wait Statement, 67
in Wait Statement, 67
to a signal, 54
Sensitivity List
Definition, 54
Examples, 52, 64
use in a finite-state machine, 65
use in a model, 53, 65
use in a test bench, 23

Propagation Delay
in Signal Assignment, 29

Range
predefined attribute (see Predefined Attribute, Range)

Record Type Declaration (see Type Declaration, Record)

Report
in Assert Statement (see Assert Statement, Report Clause)

Reserved Words
Definition, 21

Return Statement
Definition, 77
Examples, 74
use in a Function, 77
use in a Procedure, 77

Right
predefined attribute (see Predefined Attribute, Right)

Scalar Type
Definition, 77

Sensitivity
in Process Statements, 54, 67
in Wait Statement, 67

Sensitivity List
of Process Statement, 54

Sequentiality
in process statements, 23

Severity
in Assert Statement (see Assert Statement, Severity Clause)

Shift Register (see Examples, Shift Register)

Signal
sensitizing a process to, 54

Signal Assignment Statement
After Clause
Definition, 29
Implicit, 29
Conditional
Definition, 65
Examples, 64
Definition, 21, 23

Delta Delay, 29
Examples, 20, 22, 28, 30, 38, 44, 46, 52, 56, 64, 66, 74, 78, 84, 86
Inertial Delay, 29
Multiple Waveforms
Examples, 30, 38, 56, 66, 74
Propagation Delay, 29
with After Clause
Examples, 28

Signal Declaration
Examples, 22, 28, 30, 36, 38, 44, 46, 56, 64, 66, 78, 84, 86
Explicit Initialization
Definition, 31
Examples, 30, 38, 46, 64, 84
Interface (see Interface Declaration, Ports and Interface Declaration, Parameters, Signal (verify existence))

Signal Map
in Waveform File
Definition, 25

Slice Name
as target of assignment, 55
Definition, 39

Structural Description
Definition, 37

Structure
creation via Component Instantiation, 23

Subprogram (see also Function and Procedure)
Body
Definition, 75
location when Declaration is in a Package
Definition, 75
repetition of Declaration, 75
use of a Return Statement, 77
Declaration
Definition, 75
repetition in Body, 75
Number of Parameters
overloading characteristic, 75
Parameter Types
overloading characteristic, 75
Parameters (see Interface List, Subprogram Parameters)
Return Type
overloading characteristic, 75

Subtype
Natural (predefined subtype), 54

Subtype Declaration
Definition, 54
Examples, 52

Test Bench
Definition, 23

Timeout Clause
of Wait Statement (see Wait Statement, Timeout Clause)

Transaction
Definition, 25
in Waveform File, 25

Type
Bit (predefined type), 21
Bit_Vector (predefined type), 37
Definition, 21
Enumeration
Definition, 65
Integer (predefined type), 54
Natural (predefined subtype), 54
Position Number
Definition, 77
Scalar
Definition, 77

Type Conversion Function, 75

Type Declaration
Constrained Array
Examples, 22
Enumeration
Definition, 65
Examples, 64
Examples, 22, 64
Record
Examples, 22

Unconstrained Ports (see Interface List, Unconstrained)

Unit Under Test
of test bench, 23

Until
in Wait Statement (see Wait Statement, Until Clause)

Use Clause
Allowed Placement, 79
Definition, 79
Examples, 78, 84, 86

Using
a Package
with a Use Clause, 79

Utility Package (see Examples, Utility Package)

Val
predefined attribute (see Predefined Attribute, Val)

Variable
Definition, 39

Variable Assignment Statement
Definition, 39
Examples, 38, 46, 52, 66, 74

Variable Declaration
Definition, 39
Examples, 38, 46, 52, 66, 74
Explicit Initialization
Examples, 74

VSS
Actions
Assertion Violation, 53
Commands
Break After time
Motivation, 79

Wait Statement
Definition, 23
Examples, 22, 30, 38, 46, 56, 66, 74, 78, 86
Sensitivity
in Until Clause, 67
That Never Wakes
Definition, 23
Examples, 22, 30, 38, 46, 56, 66, 78, 86
Timeout Clause
Definition, 23
Examples, 22, 30, 38, 46, 56, 78
Until Clause
Definition, 67
Examples, 66, 86
signal sensitivity, 67

Wait Statements
Timeout Clause
Examples, 74

Warning
in Assert Statement (see Assert Statement, Severity Clause)

Waveform
Definition, 31
Multiple Waveform Elements, 31

When
in Case Statement (see Case Statement, When Clause)
in Exit Statement (see Exit Statement, When Clause)