User's Guide to

Accolade PeakVHDL

Professional Edition

550 Kirkland Way, Suite 200
Kirkland, WA 98033
(800) 470-2686
http://www.peakvhdl.com
email: sales@peakvhdl.com, support@peakvhdl.com

Product License, Limited Warranty and Limitation of Liability (The Small Print)

This software is licensed to you for use by only one person at a time. You may copy the software for archival purposes, but may not distribute the software to persons who are not licensed users of the product. This software is protected by U.S. and international copyright law and cannot be copied or otherwise made available to more one person at a time without violating the law. Giving or otherwise transferring all of your rights to the software to someone else will not violate copyright laws, if you give all of the software and documentation (including this license) to that person. In the event that this software is transferred to another person, you must inform Accolade Design Automation of such a transfer so that the new owner can be registered as an authorized user. Remember, once you transfer your rights to this software to another person, you cannot continue to use the software or keep any copies of the software.

Adding More Users

To allow more than one person to use this software, you must purchase additional software for each person.

Upgrades

If this software is an upgrade version of software that you previously acquired, you have not acquired two different licenses to use this software and its earlier version. This upgrade and the earlier version together constitute just one copy of the software and must be used by one person (or transferred together to only one person).

Other Limits on Your Use

Except as described in this license, you may not transfer, rent, lease, lend, copy, modify, translate, sublicense, time-share, electronically transmit or receive, or decompile or reverse engineer this software or the media and documentation.

Limited Warranty

This software package may or may not include a written guarantee, provided by Accolade Design Automation, Inc. or an independent software distributor, which for a limited period of time may entitle you to a full or partial refund of the amount actually paid for the software. Such a guarantee is subject to the terms and conditions described separately, and is not provided (expressly or implicitly) by this license.

Also, the physical media for this software product provided by Accolade Design Automation, Inc. are guaranteed to be free of defects in materials and workmanship for 120 days from the date this product was originally purchased. If a defect occurs within this 120-day period, simply return the defective media to Accolade Design Automation, Inc. and Accolade Design Automation, Inc. will replace it free of charge.

Accolade Design Automation, Inc. makes no representation or warranty regarding the content of this product, including its software and documentation. For example, Accolade Design Automation, Inc. does not warrant that the software and documentation are “error-free” or will meet the needs and requirements of a particular user. All information in the software and documentation is subject to change without notice. In addition, Accolade Design Automation, Inc. makes no representation or warranty regarding products, media, software or documentation manufactured or supplied by others.

ALL OTHER WARRANTIES, REPRESENTATIONS, CONDITIONS, EXPRESS OR IMPLIED, INCLUDING ANY IMPLIED WARRANTY OR CONDITION OF MERCHANTIBILITY OR FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED BY ACCOLADE DESIGN AUTOMATION, INC. ALL OTHER IMPLIED TERMS ARE EXCLUDED.

Limitation of Liability

The only remedy under this limited warranty or by any additional guarantee is replacement of the defective media or refund of the actual amount paid. Accolade Design Automation, Inc. disclaims any liability for damages arising from the use of this product or any other damages, including (though not limited to) lost profits or data, special, inciden-

tal, or other claims, even if Accolade Design Automation, Inc. has been specifically advised of the possibility of such claims. Regardless of the form of the claim, the only liability Accolade Design Automation, Inc. will have to you or any other person will be limited to the amount actually paid for the product.

Miscellaneous

This license and limited warranty can only be modified in writing signed by you and an authorized officer of Accolade Design Automation, Inc. If any part or provision is found to be unenforceable or void, the remainder shall be valid and enforceable. If any remedy provided is determined to have failed of its essential purpose, all limitations of liability and exclusions of damages shall remain in effect.

Use, duplication or disclosure of this software and documentation by the U.S. Government is subject to the restricted rights applicable to commercial software (under FAR 52.227-19 and DFARS 252.227-7013). Sale of this software is subject to the U.S. Commerce Department export restrictions. This software is intended for use in the country in which it is sold (or the EEC if first sold in the EEC).

This license and limited warranty shall be construed under the laws of the state of Washington, U.S.A.

You have specific legal rights under this document, and may have other rights that vary from state to state, and from country to country.

Third Edition

Printed in the U.S.A.

U0597

Acknowledgments and Copyrights

Contents

Chapter 1: What is PeakVHDL?

PeakVHDL™ is a design entry and simulation system that is intended to help you learn and use the VHDL language for advanced circuit design projects. The system includes a VHDL simulator, source code editor, hierarchy browser and on-line resources for VHDL users. If you have purchased a synthesis option, PeakVHDL allows you to control synthesis options and start the synthesis process from right within the PeakVHDL environment.

You can use PeakVHDL to create and manage new or existing VHDL projects. Because VHDL is a standard language, you can use PeakVHDL in combination with other tools (including schematic editors, high-level design tools, and other tools available from third parties) to form a complete electronic design environment.

Design Management Features

PeakVHDL includes many useful features that help you to create, modify and process your VHDL projects. The Hierarchy Browser, for example, shows you an up-to-date view of the structure of your design as you are entering it. This is

particularly useful for projects that involve multiple VHDL source files (called *modules*) and / or multiple levels of hierarchy.

The VHDL Wizards helps you create new VHDL design descriptions, by asking you a series of questions about your design requirements, and generating VHDL source file templates for you based on those requirements.

The built-in dependency ('make') features help you streamline the processing of your design for simulation and for synthesis. When you are ready to simulate your design, for example, you simply highlight the design unit (whether a source file module, entity, architecture, etc.) you wish to have processed and click a single button. There is no need to compile each VHDL source file in the design, or to keep track of your source file dependencies. PeakVHDL does it for you.

Simulation Features

PeakVHDL's built-in simulator is a complete system for the compilation and execution of VHDL design descriptions. The simulator includes a VHDL analyzer (compiler), elaborator, code generator and simulation kernel. VHDL design descriptions are compiled into a 32-bit native Windows executable form. When run, these executable files interact with the PeakVHDL system to allow interactive debugging of your circuit.

The PeakVHDL analyzer, elaborator, code generator and simulation kernel are native 32-bit Windows applications; meaning that they are fast and capable of processing very large design descriptions.

Simulation results (in the form of graphical waveforms and / or textual output) can be easily saved for use in other tools, or printed on any Windows-compatible printer.

Synthesis Features

PeakVHDL's optional synthesis packages allow design descriptions to be quickly and easily processed into netlists optimized for specific target hardware, such as FPGA devices. Synthesis options are controlled from within PeakVHDL's Options dialog box. The PeakVHDL Hierarchy Browser is used to invoke synthesis, allowing complete control over the synthesis process.

Personal Edition Feature Summary

The following features have been provided in PeakVHDL Personal Edition, and are also available in more advanced versions of PeakVHDL:

Hierarchy Browser

The Hierarchy Browser provides you with one centralized place to control the processing of your design, invoke the built-in editor to modify your design, and see how the various modules of your design are related. In addition, the Hierarchy Browser acts as a dependency checking feature, ensuring that the files associated with your project are processed in the correct order and kept up-to-date as you make modifications to them.

Source File Editor

The built-in editor allows you to quickly create and edit VHDL source files and other ASCII text files associated with your project. The editor includes useful features such as syntax coloring, automatic indenting, and global (multi-file) search capabilities.

VHDL Wizards

The VHDL Wizards help you to quickly create new VHDL modules and add them to your project. The Wizards allow you to enter the top-level specification of a design module (in the form of a port list) and automatically generates a VHDL source file template and template test bench.

VHDL Compiler and Linker

The PeakVHDL is a native-compile simulator that processes (compiles) VHDL design descriptions into Windows-compatible native x86 bject code. The VHDL compiler included with PeakVHDL Personal Edition provies support for most features of the VHDL language. (Limitations are described on the Accolade Design Automation Web site.)

The object code generated by the compiler is linked automatically to create a simulation executable compatible with PeakVHDL's SVOLE execution kernel. The PeakSIM simulation interface (see below) provides you with a graphical view of simulation results and allows you to select signals and control simulation execution.

PeakSIM Waveform Interface

The PeakSIM waveform display provides you with a graphical representation of your simulation results in a format similar to a logic analyzer. VHDL data types (including user-defined types) are dispayed in an easy-to-view format, and the interface allows you to select and order signals in the display at any time during simulation. Selectable measurement cursors make it easy to compare and measure waveform events, and a transcript window makes it easy to observe messages (such as text I/O) generated from your design as it executes.

Sample Projects

PeakVHDL Personal Edition includes dozens of sample VHDL projects to help you come up-to-speed quickly with the language and its important concepts. These projects may be copied and modified as needed as you develop your own VHDL-based projects.

Professional Edition Feature Summary

The following features have been provided in PeakVHDL Professional Edition, in addition to those features provided in PeakVHDL Personal Edition:

High-performance VHDL Analyzer and Elaborator

The Professional Edition simulator includes advanced VHDL compiler technology with faster performance and greater design capacities than are available in the Personal Edition product. The Professional Edition x86 code generator is optimized for today's high-performance x86-based processors.

Debugging Window

The source-level debugging window allows you to follow the execution of your VHDL design at the level of VHDL source file statements. This is particularly useful for debugging complex sequential statements, determining the order in which statements are processed, and understanding the impact of scheduling, delta cycles and other complex aspects of model execution.

Break Points

An important feature of the source level debugging window is the ability to set break points in your VHDL code. Break points are points at which the simulation will pause execution, making it easier for you to step through the code to analyze its execution and find errors.

VITAL (IEEE 1076.4) Support

When purchased with the PRO+VITAL option, the Professional Edition simulator product supports timing annotation using VITAL (IEEE 1076.4) compliant netlists and SDF timing data files. This feature is important for performing post-route timing simulation of FPGAs and other devices.

Command Window

The command window allows you to enter text commands, or control the execution of command files, from within the PeakSIM simulation interface. The commands available in this window give you more complete control over the simulation process, including the ability to force individual signals to specific values during debugging.

SV/OLE Programming Interface

The SV/OLE programming interface makes it possible for you to run your compiled and linked PeakVHDL projects as stand-alone Windows applications, and to interface the resulting simulation executable files directly to OLE-enabled programming environments such Visual Basic and Visual C++. This feature gives you tremendous power by allowing a form of hardware/software co-simulation. To help you get started with this powerful feature, a comprehensive on-line SV/OLE reference guide and a sample Visual Basic application are included with your PeakVHDL Professional Edition installation.

PeakLIB

The PeakLIB program supplied with PeakVHDL Professional Edition can be used to create and maintain PeakVHDL library files from your previously-compiled VHDL object files. PeakLIB is a DOS application, and is described in detail later in this manual.

PeakFPGA Integration

PeakVHDL Professional Edition is designed to integrate tightly with PeakFPGA™ for the best in FPGA synthesis. PeakFPGA is a full featured FPGA Synthesis tool that supports all major FPGA device families.

Chapter 2: Installing the Software

The PeakVHDL software follows established Windows software installation methods. The instructions in this chapter assume you will be installing the software from CD-ROM. The installation procedure for diskette distribution is similar, and is not described here.

After installing the software, but before making full use of it, you will need to register the software using a valid serial number and permanent authorization code. Follow the instructions in this chapter carefully to make sure you have correctly installed and registered the software.

System Requirements

The PeakVHDL software will install and run on any Microsoft Windows based personal computer meeting the following criteria:

- Windows 95/98 or Windows NT operating system.
- Intel (or equivalent) 486 or Pentium processor.
- 16MB (or greater) RAM.
- 17MB of available disk space (65MB with all libraries and options).

Installing Your Software

The PeakVHDL software is supplied on CD-ROM. Installation of the software is simple, and follows standard Windows installation methods.

Begin the installation by inserting the distribution CD-ROM into your CD-ROM drive unit. From My Computer or Explorer, invoke the setup utility (SETUP.EXE) found on the distribution CD-ROM.

Note: When installing, do not choose a directory that includes spaces in the path. An example of this is C:\Program Files\.

Selecting Product Options

When you begin installation, the installation software will prompt you for an installation directory and allow you to select the product features you wish to install (Figure 2-1). Choose Typical to install all files (including synthesis options) to your hard disk, or choose Compact to install only the minimum files needed for simulation. Choose Custom to remove certain product features (such as on-line manuals) from the installation if disk space is at a premium.

After installing the software on your hard disk drive, the program will create (or update) the appropriate entires in your Windows Registry, and will create a Start menu program group and items as appropriate. You then have the option to invoke PeakVHDL immediately, or you can exit the install routines and run PeakVHDL at a later time.

Note:

The PeakVHDL software does not require any modifications to your system files (such as AUTOEXEC.BAT or CONFIG.SYS).

.

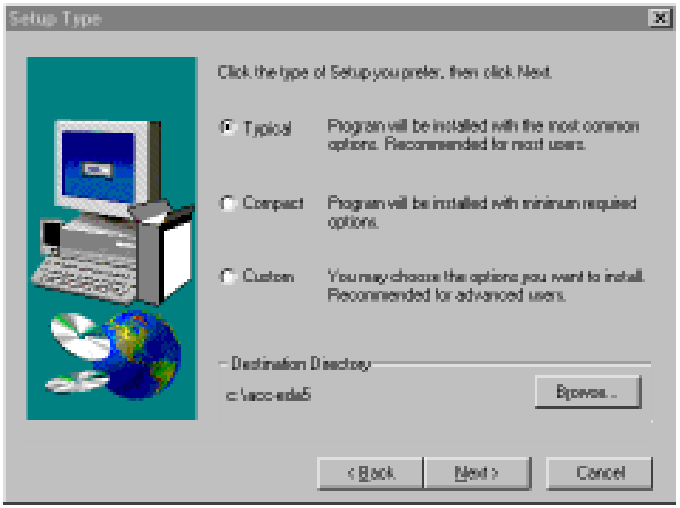


Figure 2-1: *Select Typical, Compact or Custom to install the PeakVHDL Software.*

Registering Your Software

Before beginning with PeakVHDL, you should take a moment to register the product. Although the product can be used with the Temporary Authorization Code (which has been provided for you with the software distribution), this Temporary Authorization Code will expire in as few as thirty days after you receive the software.

1. Determine Your Node ID

Before registering your software, you will need to determine the Node ID of your computer system. To determine the Node ID of your system, invoke the Display Node ID application that was installed with PeakVHDL in the **Programs / PeakVHDL** section of the Windows **Start** menu.

The Node ID is generated for your system based on certain hardware and software characteristics. There are some system upgrade or replacement situations in which your Node ID will change. If this occurs, simply contact Accolade Design Automation product support to obtain a new authorization code.

2. Obtain a Permanent Authorization Code

To permanently enable the software, you must obtain a Permanent Authorization Code from Accolade Design Automation, using the automated registration system found at the following URL: **http://www.acc-eda.com**.

Note:

If you do not have access to the World Wide Web, or are having problems accessing the automated registration system, you can obtain your authorization code direct from Accolade Design Automation. Refer to your printed distribution materials for the appropriate FAX or voice phone number. Site License users must contact Accolade Design Automation directly for a Site License Authorization Code.

When registering your software with the automated registration system, be sure to enter your name and serial number(s) correctly and completely. Your Permanent Authorization Code is tied directly to your Node ID (or site license name), and cannot be used by others. Also be sure to complete the user information section, including your complete phone number and address.

3. Enter the Authorization Code in the Software

After you have received your Personal Authorization Code, or to use temporary Authorization Code, enter the information into the Options dialog box, in the Registration tag as shown in Figure 2-2.

(You access the registration dialog by selecting the Register item from the PeakVHDL main application window.)

Enter your name, product Serial Number(s) and Personal Authorization Code in the appropriate fields. Check to make sure they are entered correctly. If you have purchased multiple product options (such as PeakVHDL simulation and PeakFPGA synthesis), enter all associated serial numbers in the Serial Number(s) field, using commas between the serial numbers as shown.

Click on the Click Here to Register button to verify the authorization code and enable the product, then click the **OK** button to exit the registration dialog.

Your PeakVHDL software is now ready for use.

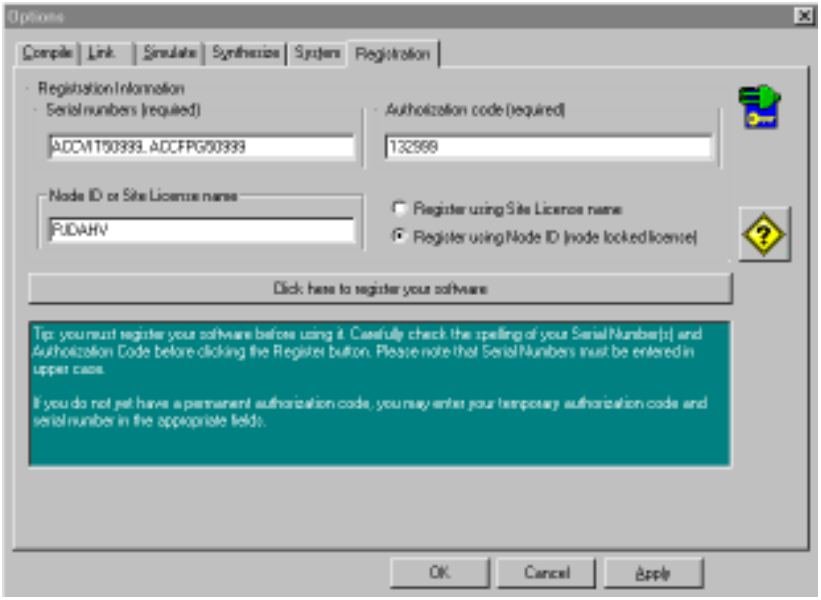


Figure 2-2: *Enter your name, serial number(s) and authorization code to enable your software.*

Chapter 3: Creating a Project

PeakVHDL operates on one or more VHDL source files that are referenced in a PeakVHDL *project file*. This chapter will describe how to create and use PeakVHDL projects, and how to create or import VHDL source files into a PeakVHDL project.

A PeakVHDL project is composed of a project file and one or more VHDL source files, which are refered to as *modules*. The project file, in addition to containing references to the various VHDL modules in your project, also includes various option settings that you have specified for the project. Each module (VHDL source file) includes one or more VHDL design units that can be selected as needed when the design is processed.

Project files (which are created with a .ACC file name extension when you select **Save Project** from the **File** menu) include information about the VHDL modules used in the design, as well as the project-specific options that you have specified. Project files do not include the actual VHDL source statements for your design; instead, the VHDL source statements are maintained in separate VHDL source files, which normally have .VHD file name extensions.

Note

PeakVHDL allows you to use alternative file name extensions, such as .VHDL or VHO, but the built-in text editor will only recognize files with a .VHD file name extension for the purpose of VHDL syntax coloring.

Creating a New Project

This and the following sections will take you step-by-step through the creation of a new PeakVHDL project, beginning with the creation of a blank project. First invoke the PeakVHDL application then, to create a new project:

1. Select **New Project** from the **File** menu, or click on the New Project icon.

A blank project will be created, and the Hierarchy Browser will appear . The Hierarchy Browser will contain references to each new VHDL module as it is added or created. Before continuing, it is a good idea to establish a working directory and project name by saving the project file. To save the project and give it a name:

2. Select **Save Project As** from the **File** menu , or click on the Save Project icon in the PeakVHDL toolbar in Figure 3-1.

After saving your project with a name, you are ready to begin creating or importing VHDL source file modules.

First, however, you may want to set a few project options.

Setting Project Options

PeakVHDL includes a variety of program options that you can specify. Some options available in PeakVHDL (such as compile flags and library paths) are related to specific projects, while others (such as the text editor font and toolbar settings) are more general and system-wide. As you learn and

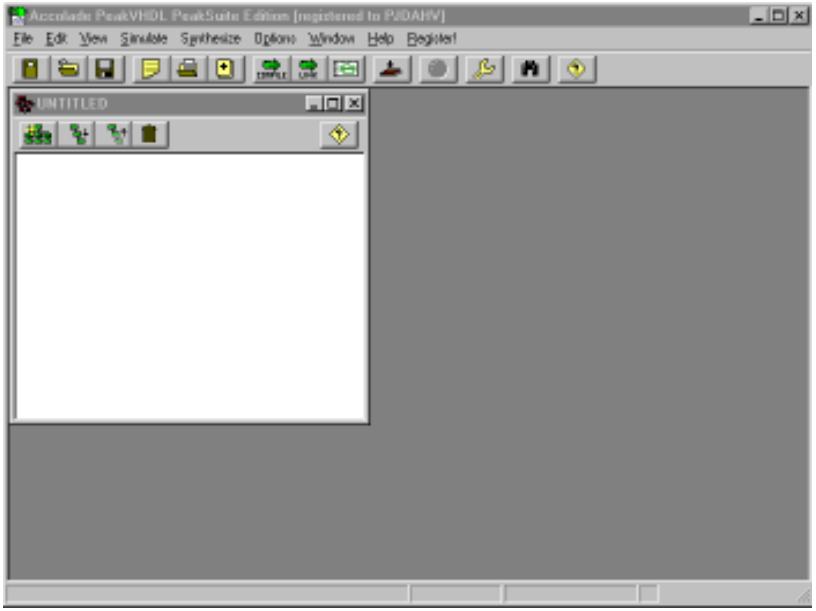


Figure 3-1*: A blank project is created, and an empty Hierarchy Browser window appears.*

use PeakVHDL's many features, you will find it useful to customize the options settings for your own preferences, and for the requirements of your projects.

To set PeakVHDL options:

1. Open the Options dialog (Figure 3-2), either by selecting any item from the **Options** menu, or by clicking on the Options icon in the PeakVHDL toolbar.
2. Use the tabbed dialog feature to select the desired option tab.

The Options dialog allows you to set a variety of options related to compilation, linking, simulation, synthesis and general program operation. (The options available to you will depend on the specific product version that you have purchased. For detailed information about available options, please consult the PeakVHDL on-line help information.)

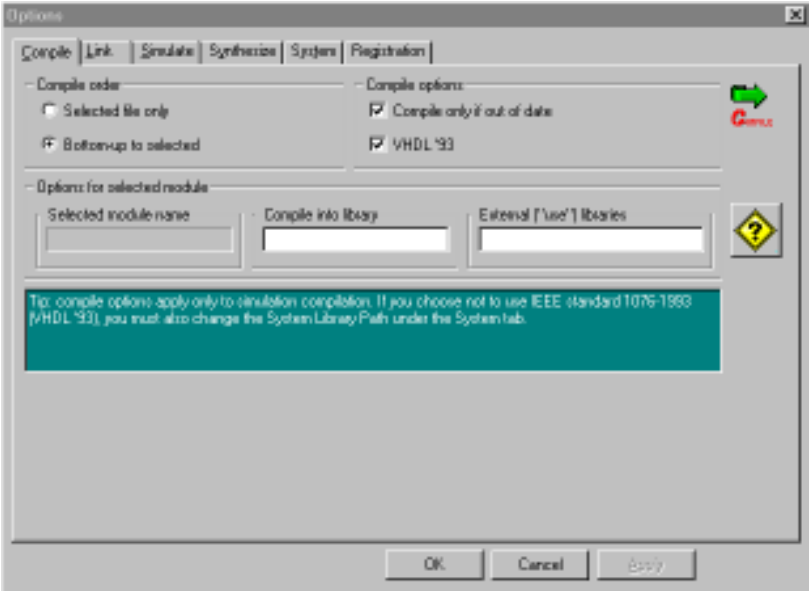


Figure 3-2: *PeakVHDL Options dialog allows you to set a variety of project options related to simulation and synthesis. PeakVHDL also includes system options so you can customize the appearance and features of your PeakVHDL installation.*

Most options that you specify in the Options dialog are saved with your project, so you can tailor the options to the requirements of a specific project. If you wish to save the options specified as the default options for all new projects, you can check the **Save options as default** option before exiting the PeakVHDL application.

When you have finished setting (or simply examining) the Options dialog:

3. Click **OK** to exit the dialog and save your new option settings, *or* click Cancel to exit the dialog without saving the new settings.

Creating a VHDL Module

There are two ways to add VHDL modules to your project, depending on whether you are building a project from existing VHDL source files or are creating a new project from scratch.

If you do not already have one or more VHDL source files to work with, you will begin by creating a new, blank VHDL module. To create a new VHDL module:

1. Select **New Module** from the **File** menu, or click on the New Module icon. The New Module dialog will appear as shown in Figure 3-3.

Note

If you have not already saved your new project, you will be prompted to save it before the New Module dialog appears.

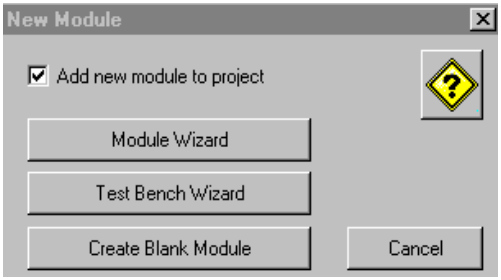


Figure 3-3: *Use the New Module button to create a new VHDL module (source file). You can create an empty module, or invoke the Module Wizard or Test Bench Wizard to create a module or test bench template.*

The New Module dialog box has three buttons that allow you to create new modules. The **Module Wizard** and **Test Bench Wizard** buttons invoke the VHDL Wizard, which is described in detail in Chapter 4. The **Create Blank Module** button adds a new, empty module to your project.

2. Click the **Create Blank Module** button to create a new, empty VHDL module.

Note

By default, the New Module dialog will add the new module to your project so it is displayed in the Hierarchy Browser. If you do not wish to have the module added to the Hierarchy Browser, you should deselect the ***Add new module to project*** *field in the New Module dialog.*

At this point, you could add some VHDL source statements to the empty module and save it (using **Save Module As** from the **File** menu). To continue this tutorial, however, you must delete the newly-created VHDL module and go on to the next section.

3. Delete the newly-created module by first closing the text editing window, then highlighting the module name in the Hierarchy Browser and selecting **Remove Module** from the **File** menu.

Note

Removing a module from the Hierarchy Browser does not remove the file from your hard disk. ***Remove Module*** *only removes the reference to the specified file from the project file.*

Adding an Existing VHDL Module

To import already-existing VHDL source files (created outside of PeakVHDL) into your project, or to copy and use a module (such as a test bench) from one of the PeakVHDL standard examples, use the **Add Module** feature instead of **New Module**. The **Add Module** feature, which is accessed by selecting **Add Module** from the **File** menu, or by clicking the Add Module icon, adds one or more VHDL source file to the Hierarchy Browser display, and to the project. To understand how this works, use the **Add Module** feature to import all the VHDL source file modules from one of the PeakVHDL standard examples:

1. Click the **Add Module** button (or select **Add Module** from the **File** menu).

2. Navigate to the PeakVHDL examples directory (typically "\acc-eda5\examples") and select one of the PeakVHDL example directories (for example, "examples\shifter", as shown in Figure 3-4).

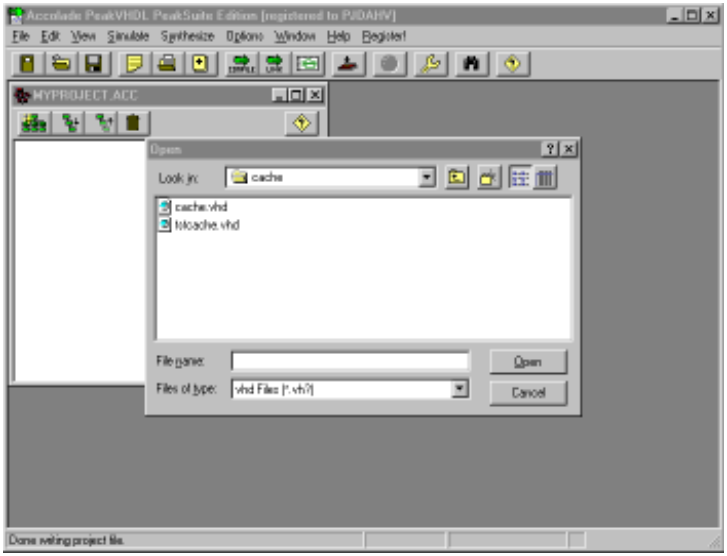


Figure 3-4: *To add one or more existing VHDL modules to the project, select the Add Module button. If the module is not in the current project directory, it will be copied.*

3. Highlight (using the shift key and mouse) all .VHD files listed.

4. Click the **Open** button to add all selected .VHD files to your project.

When you import modules using **Add Module**, if the selected VHDL source files are not in the current project directory, they will be copied to the current directory before being added to the project.

Note

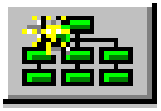
It is not possible to create projects that directly reference VHDL files located in other directories. You can, however, use the library features of PeakVHDL to create precompiled modules in different directories on your system. Refer to Chapter 8, Using PeakLIB, for more details on PeakVHDL's library features.

Examining the Project Hierarchy

When you have created or imported one or more VHDL modules for your project, you can easily examine the hierarchy of each module and the see the relationships between design units found within those modules. To examine the hierarchy of the project:

1. Make the Hierarchy Browser the active window (by clicking the mouse once within it, or on its title bar).
2. Select **Rebuild Hierarchy** from the **File** menu or use the Rebuild Hierarchy button.

When **Rebuild Hierarchy** is invoked, all modules in the project are analyzed and a hierarchy tree is created. After the tree is created, you will see small "+" icons appearing next to each VHDL module (Figure 3-5). You can use these "+" icons to push into each module and examine its contents, or you can use the **Show Hierarchy** button (shown below) to expand and view the entire project hierarchy:



When you examine the complete hierarchy for a module (either by repeatedly clicking on the "+" icons or by clicking once on the **Show Hierarchy** icon), you will see listed not only the design units that exist in the current module, but those that exist in other modules referenced from the current

***Figure 3-5**: Use the Rebuild Hierarchy button to bring the Hierarchy Browser up-to-date. You should rebuild the project hierarchy after any change to a module that might impact the project hierarchy.*

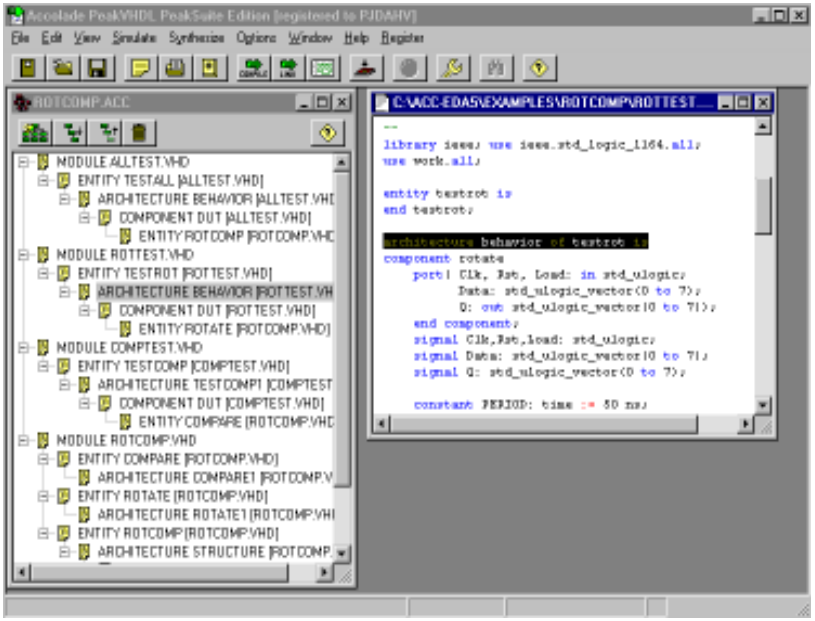


Figure 3-6: *After you select the Rebuild Hierarchy feature, the Hierarchy Browser displays the hierarchy for each module in the project. You can invoke the text editor by double-clicking on any entry in the Hierarchy Browser.*

module as well. If you wish to examine the VHDL source file associated with any design unit listed in a module's hierarchy tree, you can double-click on the design unit name and the source file will be loaded into PeakVHDL's built-in source file editor. This editor (shown in Figure 3-6) is a full-featured text editor and includes features such as search and replace, keyword coloring, and drag-and-drop editing features.

Note

If you prefer to use your own text editor, you can specify an alternate source file editor in the System Options dialog box.

Although you can edit and compile VHDL modules without first rebuilding the project hierarchy, you will not be able to link or load a module for simulation, or invoke synthesis or optimization without first bringing the project hierarchy up-to-date. In addition, you should keep in mind that the project hierarchy is not updated automatically as you modify your project. You should therefore be sure to rebuild the hierarchy

any time you make a change to the project that might impact the hierarchy of the project. Changes that can impact the hierarchy include:

- Adding or removing VHDL modules
- Changing compile library names
- Adding or removing component references
- Changing entity, architecture or component names
- Modifying references to external packages

Changing the Display Order of VHDL Modules

As you add new modules to your project, you may wish to change the order in which the modules are displayed in the Hierarchy Browser. By default, PeakVHDL adds new modules to the bottom of the Hierarchy Browser list. The order in which modules appear in the Hierarchy Browser is not significant in terms of the order of compilation, but you may want to establish a consistent standard so that you can more easily navigate in and manage your projects. For example, you may wish to place modules that are test benches at the top of the Hierarchy Browser, and maintain lower-level modules in a lower position in the list.

To move an existing module to a new position in the list, simply select that module by clicking once with the mouse, then select either **Move Module Up** or **Move Module Down** from the **File** menu.

Summary

This chapter has provided a quick introduction to the design management features of PeakVHDL. Subsequent chapters will show how you can quickly create new VHDL source modules using the Wizard features, and how you can simulate and debug your PeakVHDL projects.

Chapter 4: Using the VHDL Wizard

The VHDL Wizard is a PeakVHDL feature that allows you to quickly and easily create new VHDL modules and test benches. The VHDL Wizard prompts you to enter a list of ports (input and output signals) describing the interface to your new design module, and from that list of ports automatically generates a template module or test bench.

After the template module or test bench has been created, you can modify it to add the desired functionality and/or test stimulus.

Chapter 4 is a step-by-step tutorial designed to show you how the PeakVHDL Wizards can make the creation of new design modules fast and easy.

Before beginning this tutorial, you should create a new, empty project as described in Chapter 3.

Invoking the New Module Wizard

To create a new VHDL module using the VHDL Wizard:

1. Select the New Module from the File menu, or click on the New Module icon.
2. When the New Module dialog appears, click on the Module Wizard button as shown in Figure 4-1.

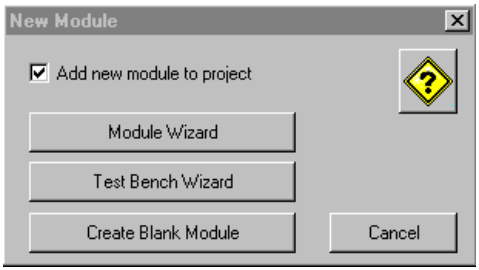


***Figure 4-1**: Click the Module Wizard button to invoke the New Module Wizard.*

Specifying the Port List

The Module Wizard generates a template VHDL source file based on the I/O specification (the *port list*) that you provide. Entering your I/O is easy: just enter the port names, one at a time, along with their direction (or *mode*, in VHDL jargon) and type. The Module Wizard helps you by providing commonly-used modes and types in drop-down selection lists, and by checking to make sure the names that you enter are valid VHDL identifiers.

For this tutorial example, we will create a simple shift register that accepts 8-bit data, and shifts (rotates) this data one bit position on the next rising edge of the clock. To describe the top-level entity and interface to this sample design in the Module Wizard:

1. Enter the name of the new module (its VHDL entity name) in the Entity Name field.
2. Enter the name of the new module's architecture in the Architecture Name field, or simply leave the field with its default value (architecture name **behavior**).

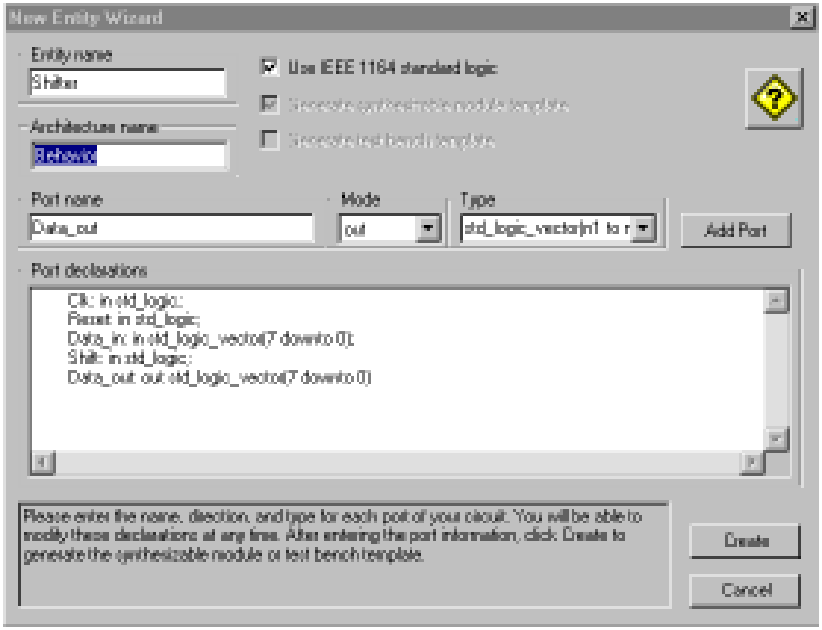


Figure 4-2: *Use the Module Wizard to quickly describe the interface to your new module. Each port of the module is entered with a name, direction and type. You can edit the port list at any time before clicking the Create button.*

3. Use the Port Name, Mode and Type fields to add port declarations for each of the inputs shown in Figure 4-2. Be sure to select the correct mode (**in** or **out**) for each port as shown. Click the Add Port button to add each port to the port declarations list.

As you enter the ports, you can make changes to them at any time by clicking in the Port declarations edit window. For example, you will probably want to edit ports that are array types to give them valid ranges as shown.

When modifying items within the Port Declarations window, keep the following rules in mind:

- Each entry in the port list, with the exception of the last entry, must be terminated by a semicolon;
- There must be only one entry (port identifier) on each line; do not attempt to combine multiple port names on a single line.

- If you make use of IEEE standard logic data types (including std_logic, std_ulogic, std_logic_vector and std_ulogic_vector), you must make sure the **Use IEEE standard logic** check box is selected.

After you have entered all the ports for your design (and have verified that they have the desired modes and types), you are ready to create the new module and save it to a file. To do this:

4. Click the **Create** button in the Module Wizard dialog.

When you click **Create**, the Module Wizard prompts you for a file name (typically a .VHD file):

5. Enter a file name (such as SHIFTER.VHD) or accept the default file name.

PeakVHDL will save your new module to the specified file and add a reference to the file to your project as shown in Figure 4-3.

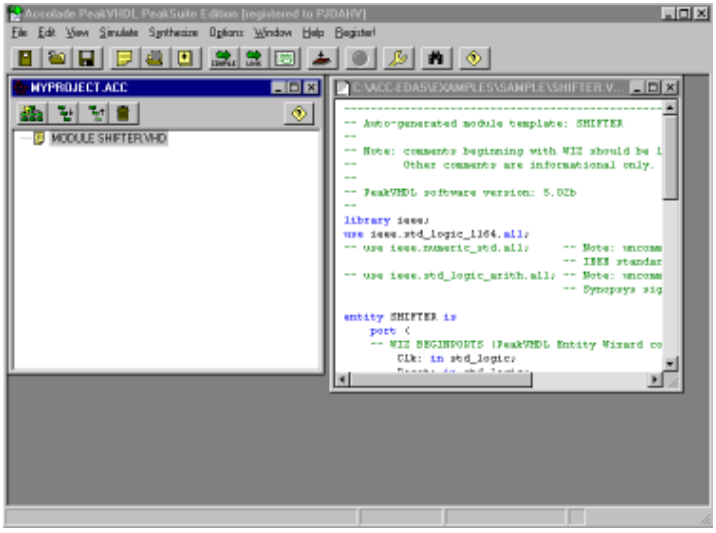


Figure 4-3 *When you click the Create button in the Module Wizard dialog, PeakVHDL creates a new template module for you, and adds the new module to your project. You can begin editing your new module immediately.*

Adding Functionality to Your New Module

After PeakVHDL's Module Wizard has created your new module, you will need to edit the module to add the appropriate functionality. While PeakVHDL can't read your mind to know what the intended function of your new module is, it does make the process easier by generating sample code, and by inserting comments to help guide you as you modify your VHDL code.

Because most new VHDL modules you create will include at least one registered element, the Module Wizard inserts sample VHDL code and comments showing you how to write a synthesizable register element, with a suggested (synthesizable) style for describing the clock and reset logic. To give you an example, the following VHDL source code was generated by the Module Wizard from the port specifications described in the previous section:

```
------------------------------------------------------------------
-- Auto-generated module template: SHIFTER
--
-- Note: comments beginning with WIZ should be left intact.
--       Other comments are informational only.
--
library ieee;
use ieee.std_logic_1164.all;
-- use ieee.numeric_std.all;
-- use ieee.std_logic_arith.all;

entity SHIFTER is
   port (
   -- WIZ BEGINPORTS (PeakVHDL Entity Wizard command)
      Clk: in std_logic;
      Reset: in std_logic;
      Data_in: in std_logic_vector(7 downto 0);
      Shift: in std_logic;
      Data_out: out std_logic_vector(7 downto 0)
   -- WIZ ENDPORTS (PeakVHDL Entity Wizard command)
   );

end SHIFTER;
```

```
architecture BEHAVIOR of SHIFTER is
-- Note: signals, components and other objects may be
-- declared here if needed.
begin

   -- Sample clocked process (synthesizable) for use in
   -- registered designs.
    --
   -- Note: replace _RESET_ and _CLOCK_ with your reset
   -- and clock
   -- names as appropriate. (Delete this process if the design
   -- is not registered.)
  P1: process(_RESET_, _CLOCK_ )
   -- Note: variables may be declared here if needed.
   begin
      if _RESET_ = '1' then
         -- Registers are reset here. Be sure you include
         -- reset values for all signals that are assigned
          -- logic in the process.
      elsif rising_edge(_CLOCK_) then
         -- Note: registered assignments go here. Remember
         -- that signal assignments do not take effect until the
          --process completes.
      end if;
   end process P1;

   -- Note: concurrent statements (including concurrent assignments
   -- and component instantiations) go here.

end BEHAVIOR;
```

This template source code can be quickly and easily modified to described the desired function (a shifter). The exact changes needed for this template source code are:

1. The template's "dummy" clock and reset signals (**_CLOCK_** and **_RESET_**) must be replaced with the actual clock and reset lines for the design (**Clk** and **Reset**, in this example).
2. Reset assignments must be added (after the first **if** statement).

3. The clocked operation of the design must be described, using whatever VHDL statements are appropriate.

4. Concurrent statements (such as combinational assignments or component instantiations, if any) must be entered where indicated.

Continuing with the shifter example, we might modify the template source code to describe our shifter as follows:

```
------------------------------------------------------------------
-- Auto-generated module template: SHIFTER
--
-- Note: comments beginning with WIZ should be left intact.
--       Other comments are informational only.
--
library ieee;
use ieee.std_logic_1164.all;
-- use ieee.numeric_std.all;
-- use ieee.std_logic_arith.all;

entity SHIFTER is
   port (
   -- WIZ BEGINPORTS (PeakVHDL Entity Wizard command)
      Clk: in std_logic;
      Reset: in std_logic;
      Data_in: in std_logic_vector(7 downto 0);
      Shift: in std_logic;
      Data_out: out std_logic_vector(7 downto 0)
   -- WIZ ENDPORTS (PeakVHDL Entity Wizard command)
   );

end SHIFTER;

architecture BEHAVIOR of SHIFTER is
-- Note: signals, components and other objects may be
-- declared here if needed.
begin

   -- Sample clocked process (synthesizable) for use in
   -- registered designs.
```

```
P1: process(Reset, Clk )
begin
    if Reset = '1' then
        -- Registers are reset here.
        Data_out <= (others => '0');
    elsif rising_edge(Clk) then
        -- Note: registered assignments go here.
        if Shift = '1' then
            Data_out <= Data_in(6 downto 0) & Data_in(7);
        else
            Data_out  <= Data_in;
        end if;
    end if;
 end process P1;

end BEHAVIOR;
```

For more complex (and realistic) designs, you will need to make many such changes and additions to acheive the desired functionality.

Compiling the New Module

After you have modified your new module, you will need to compile it to verify that you have entered your VHDL statements correctly. To compile the file:

1. Select your new module by highlighting its entry in the Hierarchy Browser as shown in Figure 4-4.

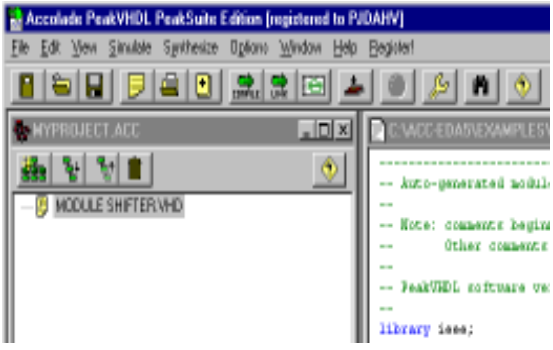


Figure 4-4: *To compile a VHDL Module, highlight the module in the hierarchy browser window and click the compile button.*

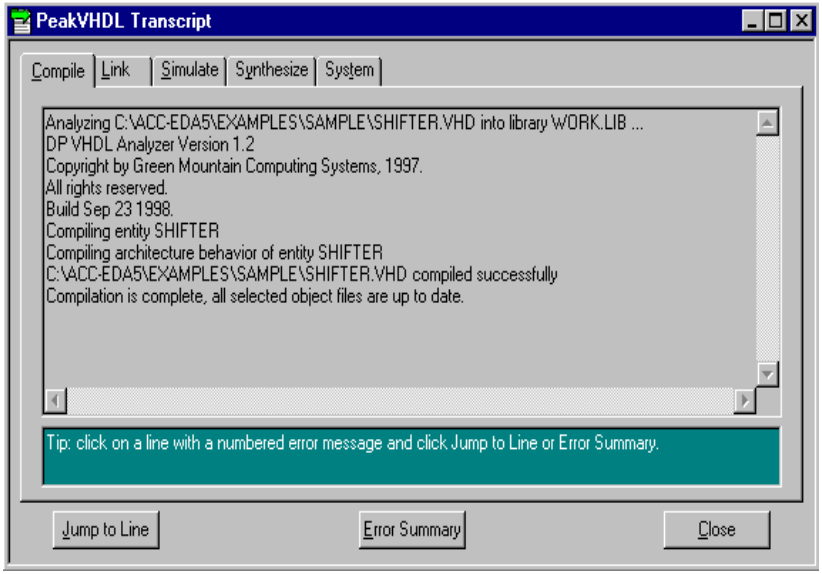


Figure 4-5: *During compilation, error and status messages are written to the transcript window.*

2. Click the Compile button, or select **Compile Selected** from the **Compile** menu.

After you have invoked the compiler, a transcript window will appear in which any error messages will be displayed (Figure 4-5).

Note

Depending on the number of syntax errors you have introduced during your editing session, you may need to compile the module more than once. Refer to the Using Simulation chapter for more details about finding and fixing syntax and other errors.

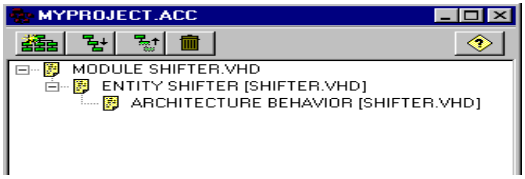


Figure 4-6: *You can click the Rebuild Hierarchy button at any time to update the project Hierarchy.*

Updating (Rebuilding) Your Project Hierarchy

After you have succesfully compiled your new module, and any time you make a change to it (or any other file) that might impact the hierarchy of your design, it is important to update the information in the Hierarchy Browser by rebuilding the project:

1. Rebuild the hierarchy by clicking the Rebuild Hierarchy Button as shown in Figure 4-6.

After you have rebuilt the project hierarchy, you can view the hierarchy for your new module by clicking on the small plus sign icon to the left of the module name, or by clicking the Show Hierarchy Button to expand the hierarchy for the entire project.

Using the Test Bench Wizard

Before processing a module for simulation, you will need to provide PeakVHDL with a test bench. Test benches are VHDL modules that provide input stimulus (and, if desired, output value checking) for VHDL modules that are to be simulated. There are many ways to write test benches (and you can examine many different test bench styles by perusing the PeakVHDL examples directory), but nearly all VHDL test benches have the following in common:

- They have an entity declaration with no input or output ports.
- They have one or more component declarations describing the interface to the VHDL module being tested (called the *unit under test*, or *UUT*).
- They have a series of signal declarations defining local (top-level) signals onto which input values can be assigned, or from which output values can be observed.

- They have one or more component instantiations corresponding to the module(s) being tested. These component instantiations connect the local signals of the test bench to the corresponding ports of the unit under test (UUT).
- They have one or more **process** statements describing the sequence of inputs applied to the UUT, and the tests to be performed (if any) on the UUT outputs.

Writing a test bench that includes all of these elements is not difficult, but it can be tedious if the module being tested has many input and output ports.

The Test Bench Wizard helps by automatically generating a basic framework of a test bench, and helps to reduce typing errors by automatically filling in such things as port lists, component declarations and component instantiations. The Test Bench Wizard also generates a sample process for a background clock, and generates informative comments that guide you as you develop your test stimulus.

Invoking the Test Bench Wizard

To create a new VHDL test bench using the Test Bench Wizard:

1. Select (by highlighting) the VHDL module that this test bench will be referencing. For this example, select module SHIFTER.VHD.

For a design with multiple VHDL modules, you would select the top-level module in your project's hierarchy, unless you are creating a test bench that is intended to test only one component of the design.

2. With the module to be tested highlighted in the Hierarchy Browser, select the **New Module** item from the **File** menu, or click on the New Module icon.
3. When the New Module dialog appears, click on the Test Bench Wizard Button as shown in Figure 4-7.

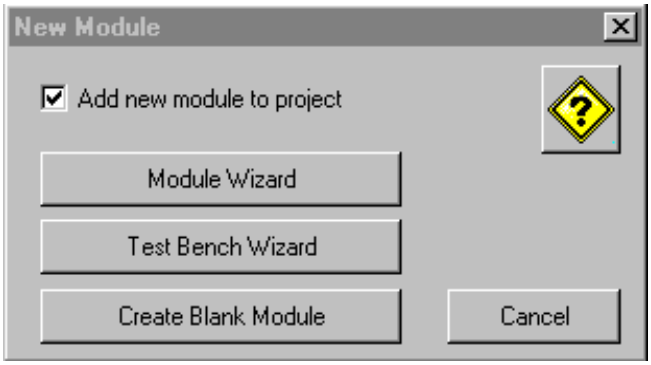


Figure 4-7: *The Test Bench Wizard is accessed from the New Module dialog. Be sure to highlight the module to be tested before invoking the Test Bench Wizard.*

Verifying the Port List

When the test bench wizard is invoked it examines the port list of the module that you have selected (in this case SHIFTER.VHD) and attempts to fill in the port declarations edit box for you, as shown in Figure 4-8. All you need to do is verify that all of the ports have been listed along with their correct direction and type:

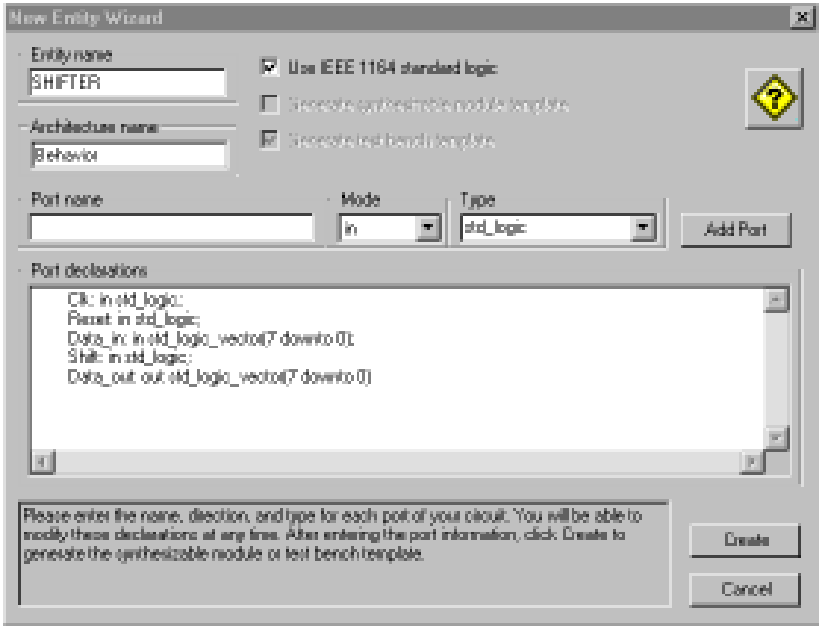


Figure 4-8: *The test Bench Wizard attempts to read your VHDL module and automatically fill in the Port declarations edit window. All you need to do is verify the port declarations and click Create.*

1. Examine the port declarations to ensure they match the declarations show, then click Create.

Note:

If you are generating a test bench for a module that was not created using the module wizard, you may need to manually enter the port list. You can save time by using cut and paste to paste in text copied from an editor window.

2. When prompted, enter a name for the new test bench module, or accept the default name (in this case TEST_SHIFTER.VHD).

Your new test bench module is now complete, and is displayed as shown in Figure 4-9.

Modifying the Test Bench

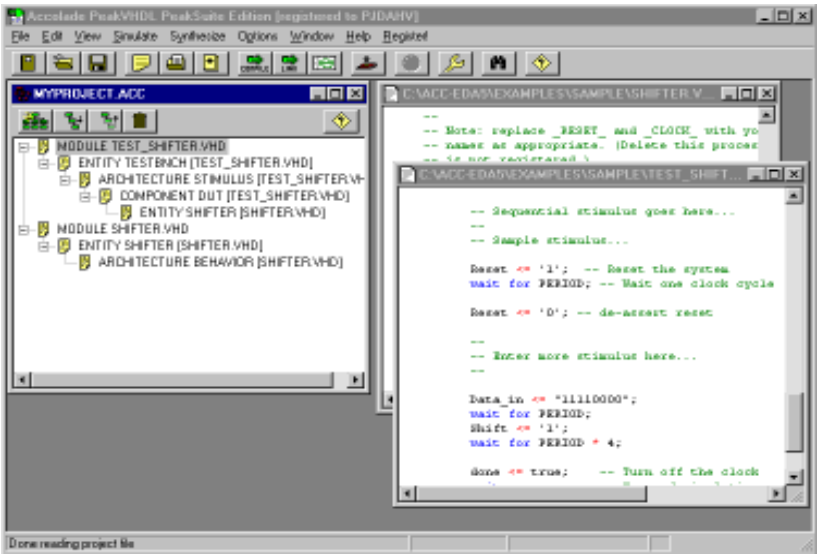
This test bench template source code must now be modified to describe the desired test stimulus. The exact changes needed to this template will depend on how extensively you want to test the design. We can create a simple test sequence for this shifter by doing making the following modifications to the source code:

1. In process **Clock1**, replace the template's "dummy" clock signal (**_CLOCK_**) with the actual system clock signal **Clk**.

2. In process **Stimulus1**, replace the assignments to **_RESET_** so they instead refer to signal **Reset**.

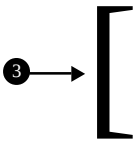
3. Add some additional stimulus to this design as shown in the source file listing that follows. The sample stimulus assigns a value to signal **data_in** , then sets the **shift** input to '1'. A subsequent wait statement will cause the simulation to move forward for some period of simulated time (in this case 100ns). Similar sequences of assignments and **wait** statements apply additional test inputs.

Figure 4-9: *The new test bench template is generated and added to the project.*




```
CLOCK1: process
   variable clktmp: std_ulogic := '0';
begin
   wait for PERIOD/2;
   clktmp := not clktmp;
(1)→  Clk <= clktmp;  — Attach your clock here
   if done = true then
     wait;
   end if;
end process CLOCK1;

STIMULUS1: process
begin

   — Sequential stimulus goes here...
   —
   — Sample stimulus...

(2)→  Reset <= '1';  — Reset the system
   wait for PERIOD; — Wait one clock cycle

(2)→  Reset <= '0'; — de-assert reset

   —
```

```
        — Enter more stimulus here...
        —

        Data_in <= “11110000”;
        wait for PERIOD;
        Shift <= ‘1’;
        wait for PERIOD * 4;

        done <= true;    — Turn off the clock
        wait;            — Suspend simulation
    end process STIMULUS1;

end stimulus;
```



After you have made the above changes to your test bench template, save the module and compile it:

4. Highlight the test bench module in the Hierarchy Browser and click the Compile button.

5. After you have successfully compiled your new test bench, click the Rebuild Hierarchy Button to bring the Hierarchy Browser display up-to-date as shown in Figure 4-11.

Your project is now ready for simulation.

Summary

This chapter has described the basic features of the PeakVHDL Module and Test Bench Wizards. You will find the PeakVHDL Wizards to be a big time-saver as you enter and verify new design modules. In the next chapter, you’ll learn how to link and load PeakVHDL projects for simulation.

Chapter 5: Using Simulation

The previous two chapters described how to create new projects and add or create VHDL source file modules. This chapter will describe how you can use PeakVHDL's built-in simulator features to verify your VHDL design projects.

Understanding Simulation

Simulation of a VHDL design description using PeakVHDL involves three major steps:

- Compiling the VHDL modules into an intermediate object file format.
- Linking the object files to create a simulation executable.
- Loading the simulation executable and starting the simulation.

Each of these three steps is represented by an icon button in the PeakVHDL application. If the dependency features of the application are enabled, the PeakVHDL application will check the date and time stamps of files, and will examine the hierarchy of your design to determine which files must be compiled

and linked at each step. When a simulation executable has been successfully linked and loaded, the Waveform Display appears and you are ready to start a simulation run.

To help you understand this process, we will load and simulate the sample project developed in the previous chapter.

Note

If you did not follow the tutorial in the previous chapter to create a new project, you can follow these steps using one of the standard examples provided with PeakVHDL.

Loading the Sample Project

To load the sample project:

1. Invoke the PeakVHDL application and select **Open Project** from the **File** menu. Navigate to the **Examples\Shifter** directory (or to your project directory created in the previous tutorial) and choose the **Shifter.ACC** file. The Project will be loaded as shown in Figure 5-1.

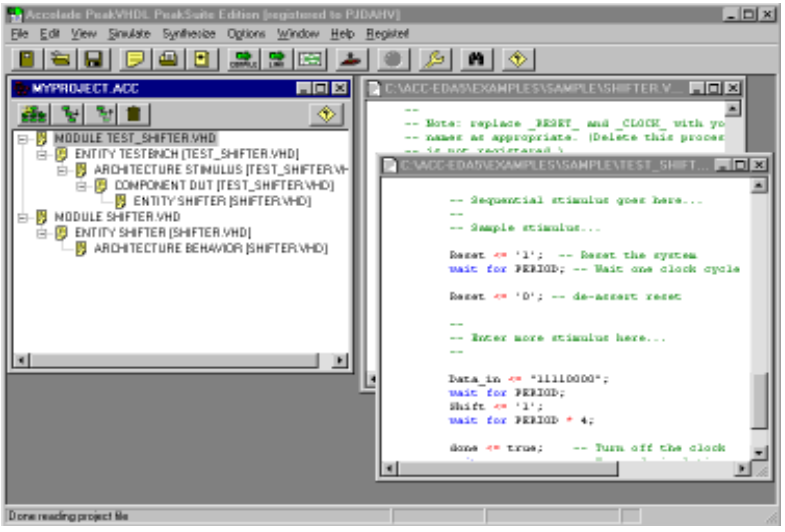


Figure 5-1: *With a project loaded into the PeakVHDL application, the Hierarchy Browser becomes your primary view of the files and designs units making up the project.*

Using the Hierarchy Browser

After you open the project, you will see that there are two modules listed in the Hierarchy Browser (Figure 5-2). These modules (TESTSHIF and SHIFTER) are VHDL modules that were entered to describe the operation of the sample circuit. TESTSHIF is a test bench for the circuit, while SHIFTER describes the function of the shifter circuit itself. You can examine or modify either of these modules by double-clicking on them to invoke a Source Code Editor window as described in the previous chapter.

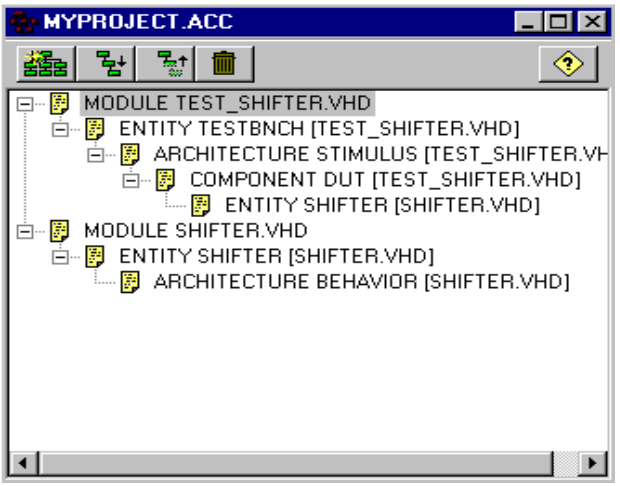


***Figure 5-2**: Use the plus and minus icons to examine the hierarchy for each module.*

The Hierarchy Browser does not provide any immediate indication of which module represents the “top” of your design. (The order of modules appearing in the Hierarchy Browser is not significant.)

You may choose to select different top-level modules depending on whether you are invoking simulation or synthesis, and depending on whether you want to simulate just a portion of the circuit or simulate the entire circuit. You may also have more than one top-level test bench in your project.

You can, however, display the hierarchy and file dependencies for any module displayed in the Hierarchy Browser. By clicking on the small white "+" icons to the left of each module, you can view the hierarchy for that module.

The Hierarchy Browser is the point from which you initiate all processing of your design, from compiling and linking to synthesis and simulation.

Compiling Modules for Simulation

Setting Compile Options

Before compiling this design, take a moment to examine the compile options that have been selected:

1. Highlight the module SHIFTER in the Hierarchy Browser.
2. Select **Compile** from the **Options** menu (or click on the Options button) to bring up the Compile Options dialog. The options should be set as shown in Figure 5-3.

The options set are:

- **Bottom up to selected**. This option tells the compiler to examine the dependencies of the project, and to compile lower-level VHDL modules before compiling higher-level modules that depend upon them.
- **Compile only if out of date**. This option enables the date and time stamp checking features so that modules are not compiled unless they are out of date. This can save time when you are compiling a large project repeatedly (such as when fixing syntax errors in higher-level modules).
- **Compile Into Library**. This option specifies that the current module, SHIFTER, is to be compiled into a named library, in this case WORK.

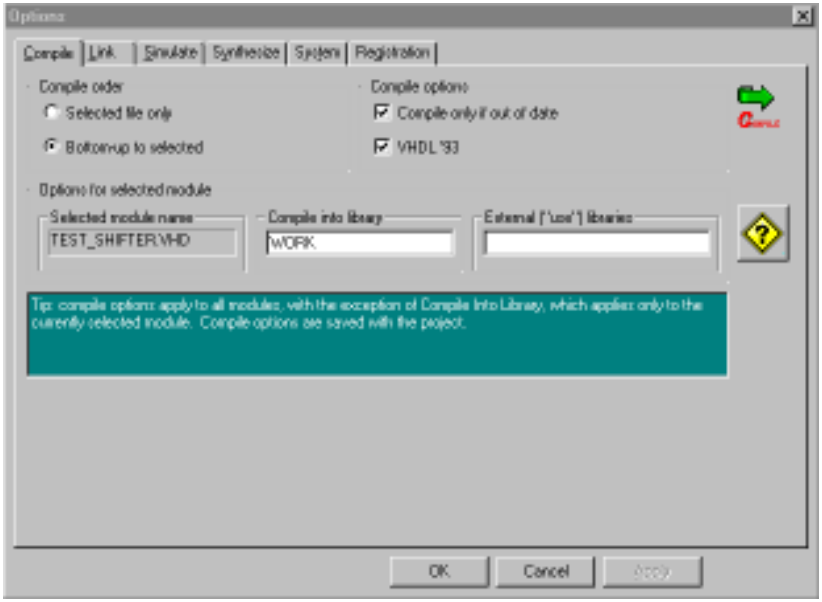


Figure 5-3*: The Options dialog allows you to set processing options for compiling, linking and simulation.*

(For detailed information about these and other options, please consult the PeakVHDL on-line help information.) When you have verified that the options are set to these values, select the Close button to close the Options dialog.

Starting a Compile

The next step is to compile the source modules. With the **Bottom up to selected** option selected, you have two options:

- You can first compile the SHIFTER module, then compile the TESTSHIF module ***or***,
- You can simply compile the TESTSHIF module, and let the dependency features automatically compile the lower-level SHIFTER module.

Use the second method to compile the two source files:

1. To start the compile, highlight the TESTSHIF module in the Hierarchy Browser and click the Compile button.

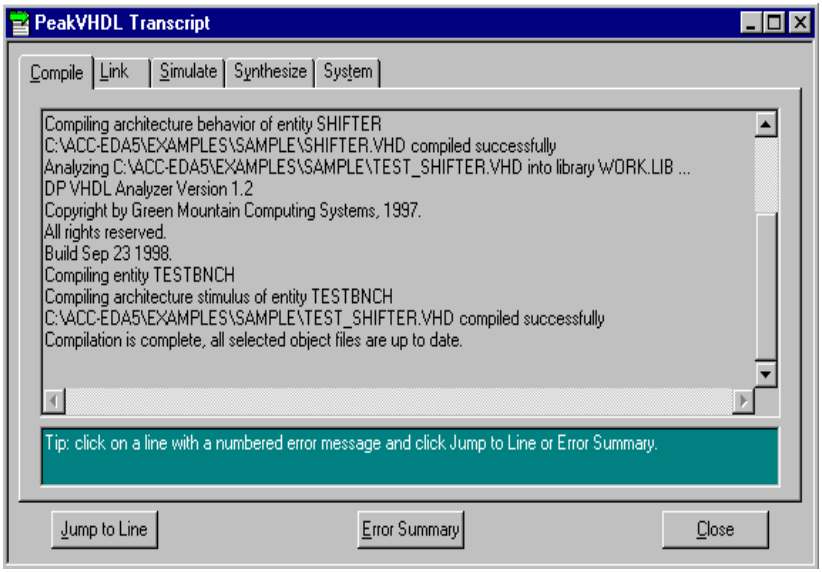


Figure 5-4: *During processing, status and other messages are displayed in a scrollable transcript window. If you wish, you can save the contents of the window or print it to any Windows compatible printer.*

During compilation, status and error messages are written to the Transcript window (Figure 5-4). If you wish, you can save these messages to a file or print them directly.

When compiled, each VHDL source file (module) in the project is processed to create an intermediate output file (an *object file*). These files, which have a .O file name extension, must be linked together to form a *simulation executable*.

Note:

If system or other errors occur during the processing of this design example, you should check to make sure you have properly installed and registered the PeakVHDL software. The software will not operate without first being registered.

Linking Modules for Simulation

Setting Link Options

Before linking the modules, take a moment to examine the link options:

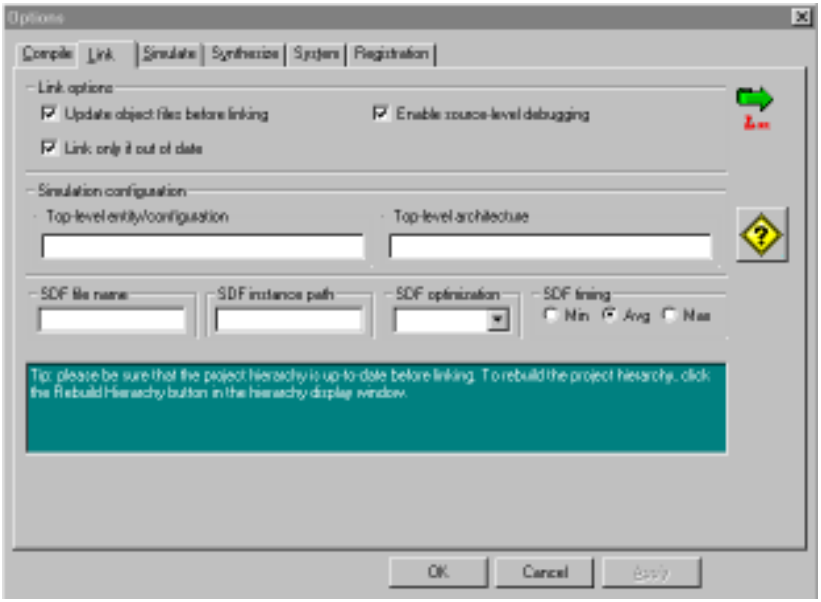


Figure 5-5: *The link options allow you to specify a default top-level entity and architecture, and also give you control over the automatic update features of the interface.*

1. Highlight the module TESTSHIF in the Hierarchy Browser.
2. Select **Link** from the **Options** menu (or click on the options button and choose the Link tab) bring up the Link Options dialog. The options should be set as shown in Figure 5-5.

The options set are:

- **Update object files before linking**. This option tells the linker to examine the dependencies of the project, and to compile lower-level VHDL modules before linking. (If the **Compile only if out of date** option is specified in the Compile Options dialog, only those source files that are out of date will be recompiled.)

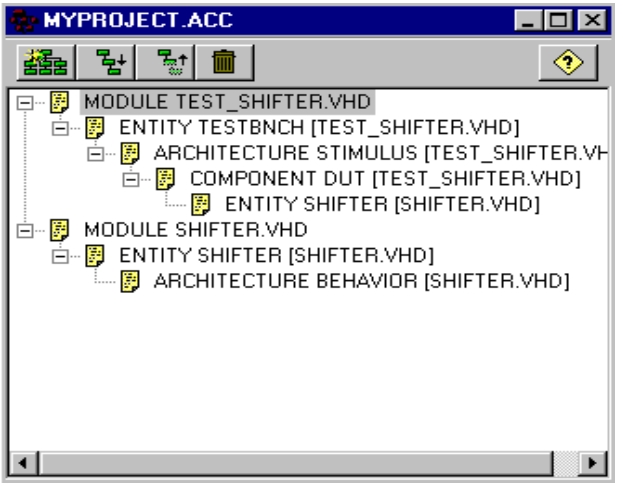


Figure 5-6: *To link the project for simulation, you must first select a top-level module, entity or architecture.*

- **Link only if out of date**. This option enables the date and time stamp checking features so that the modules are not re-linked unless the simulation executable is out of date.
- **Design Unit Selected**. These fields allow you to specify a default top-level entity and architecture for the selected module. Because this project does not include multiple entities and architectures within the top-level module, you can leave these fields blank.

When you have verified that the options are set to the values shown, select the Close button to close the Options dialog.

Starting a Link Operation

When linking a project, the PeakVHDL linker collects all object files required for the selected top-level module, and combines these object files with any libraries you have specified in your design (such as the IEEE standard logic library) to create the simulation executable.

To link your design and create a simulation executable:

3. Highlight the TESTSHIF module in the Hierarchy Browser (Figure 5-6).

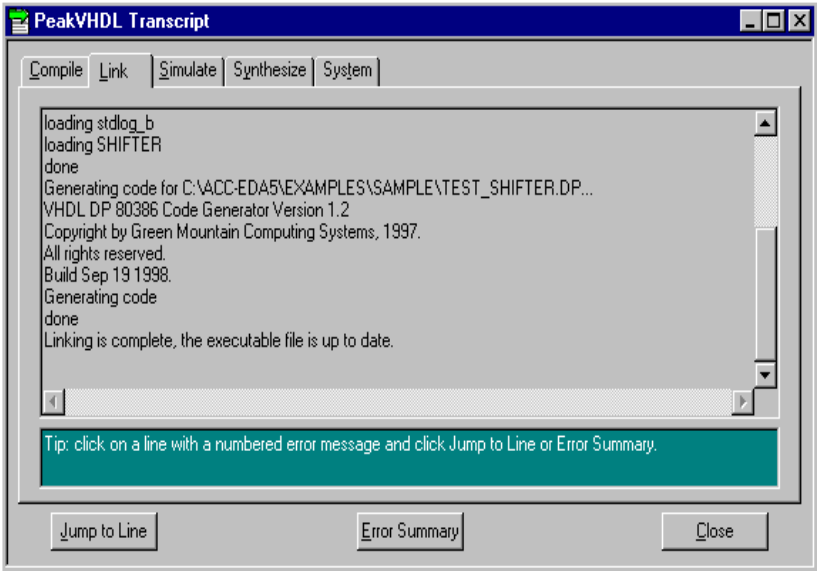


Figure 5-7: *During the link process, the object files (created as a result of compilation) are combined with any external libraries to create a simulation executable.*

4. Click on the Link Button to initiate the link operation.

During the linking process, messages will be written to the PeakVHDL transcript as shown in Figure 5-7.

The result of linking is a simulation executable file ready to be loaded for simulation.

Setting Simulation Options

Before loading the simulation executable, take a moment to examine the simulation options:

1. Make sure the TESTSHIF module is still highlighted in the Hierarchy Browser.
2. Select **Simulate** from the **Options** menu (or click on the options button and choose the Simulate tab) to bring up the Simulate Options dialog. The options should be set as shown in Figure 5-8.

The options set are:

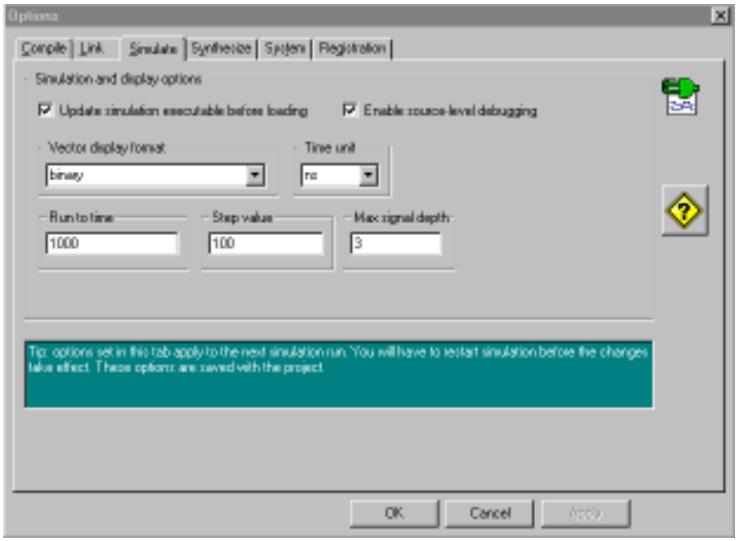


Figure 5-8: *Simulation options include waveform display format, and default run-to and step times.*

- **Update simulation executable before loading**. This option tells the simulator to examine the dependencies of the project and, if necessary, compile lower-level VHDL modules and re-link them before loading.
- **Vector Display Format**. This option specifies how vector (array) data types should be displayed. Select **binary** for this field.
- **Run to Time and Step Value**. These fields allow you to specify a default amount of time that the simulation should run. These values can be changed during simulation if necessary.
- **Time Unit**. This field specifies the unit of time (eg. ns, ps) to be used during simulation.

(For detailed information about these and other options, please consult the PeakVHDL on-line help information.) When you have verified that the options are set to the values shown:

3. Select the Close button to close the Options dialog.

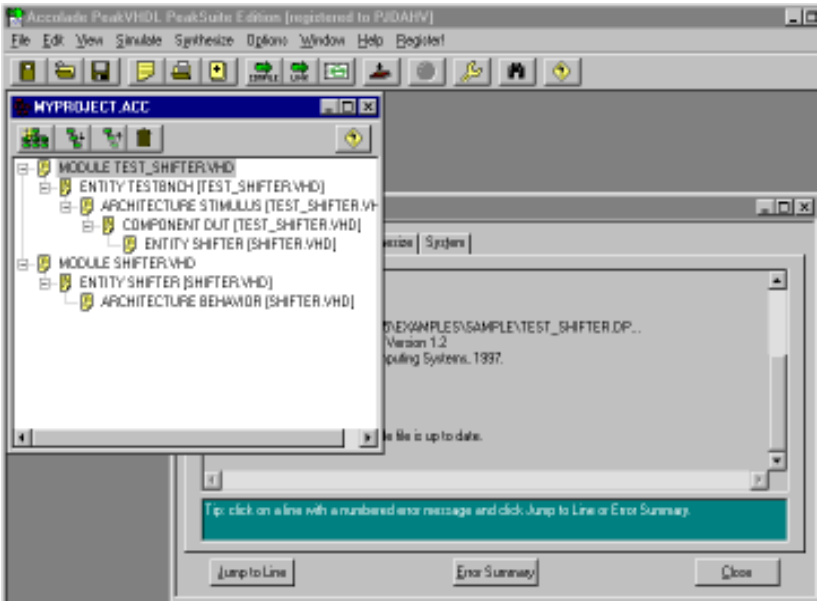


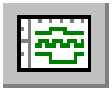
Figure 5-9: *You must select a test bench module when loading simulation.*

Loading the Simulation Executable

At this point, you have compiled each of the VHDL modules into an object file format, and have linked all of the object files to form a simulation executable. To start the simulation process, you will use the Load Simulation Button.

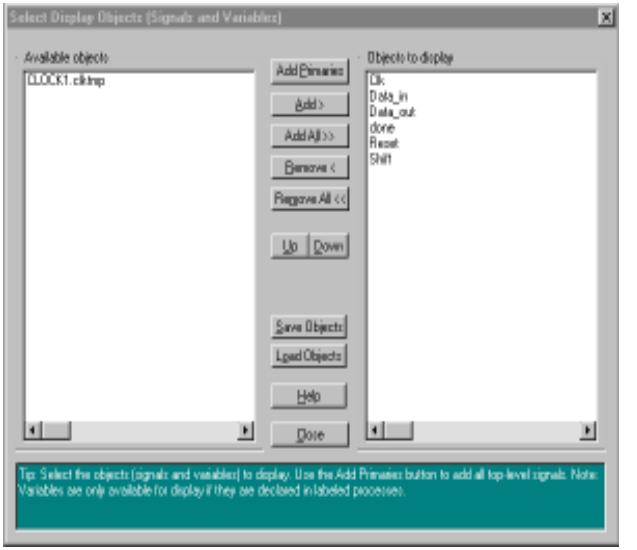
To load the simulation executable:

1. Make sure the TESTSHIF module is selected in the Hierarchy Browser (Figure 5-9) and click on the Load Button shown below:



The simulation executable will be loaded and the PeakVHDL simulation application (PeakSIM) will appear.

Figure 5-10: *The Select Display Objects dialog allows you to select specific signals withing your design for display. You can save your selections to a file using the Save Objects button.*



Selecting Signals to Display

When you click **Load Button**, the PeakSIM application appears and immediately displays a Select Display Objects dialog. This dialog allows you to choose signals to observe during simulation. If you select the Add Primaries Button (the top-most button), all of the top-level signals in your design will be moved to the display window.

You can select other signals within your design to probe important signals in your design.

Note:

You can add as many signals as you wish from the Signal Display window, but the speed of the PeakSIM application will be negatively impacted if you select too many signals. For this reason you should select only those signals that are important for verification and debugging of your design.

To select signals to observe in simulation:

1. Use the Add Primaries button to move the top-level signals from the Available window to the Displayed window.

2. Highlight individual signals in the Available window and use the Add button to move individual signals to the Displayed window.

3. Use the Up and Down buttons to rearrange signals, putting them in any display order you wish.

Figure 5-10 shows the Select Display Objects dialog with some of the design's signals selected for display.

4. When you are satisfied with the selected signals, click the Close button to close the dialog and prepare the simulation.

Changing Simulation Options

You are now ready to simulate the sample project. Before doing so, however, you may want to modify some of the simulation options that were specified earlier for the project. You can do this by choosing the Options menu item, or by clicking on the Simulation Options button in the PeakSIM toolbar.

The options that can be changed in the PeakSIM options dialog window are shown in Figure 5-11. Many of these options are the same options that you specified previously in

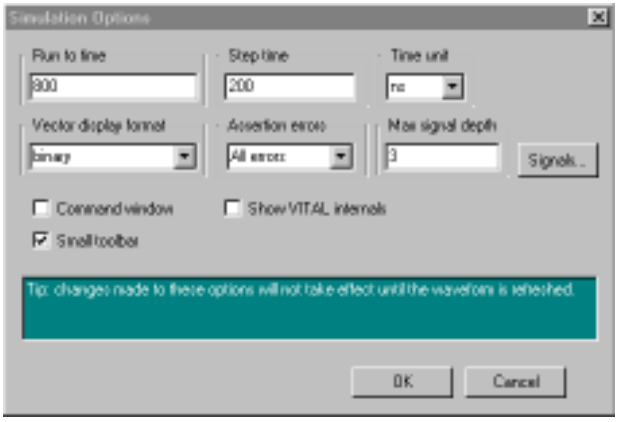


Figure 5-11: *Change the simulation end time and other options by using the Simulation Options dialog. Note also that the initial (default) values of many of these options are specified in the PeakVHDL Simulation Options dialog described previously.*

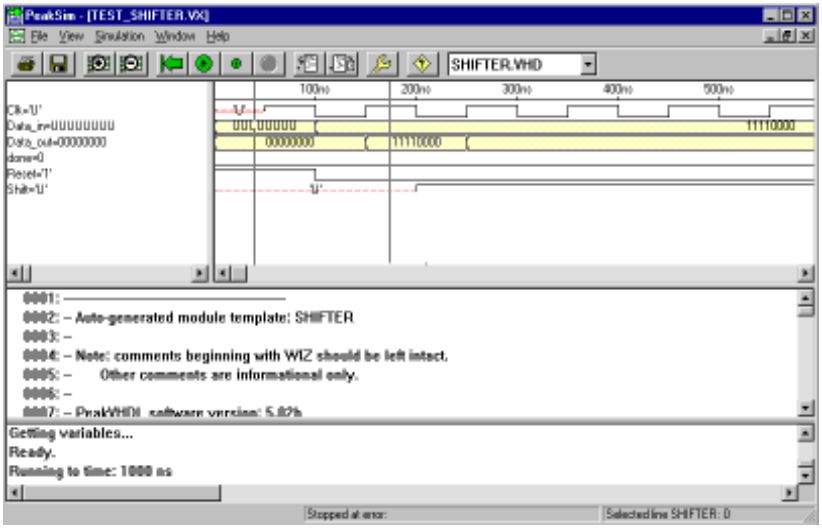


Figure 5-12: *When you click the GO button, the simulation advances to the end time you have specified. The waveform display can be zoomed and panned to display all or part of the waveform.*

the PeakVHDL Options dialog. These options are repeated here to allow you more control over the simulation as you debug your design. For example, you can update the simulation time to reflect the desired simulation end time, or temporarily choose an alternate vector display format.

For detailed information about available options, please consult the PeakSIM on-line help information.

Starting a Simulation Run

After you have selected signals to observe during simulation, you can click on the GO Button to start the simulation.

Simulation will run until either:

- The specified simulation end time (duration) has been reached or,
- All processes in your project have suspended.

To start simulation using the previously-specified run time:

1. Click on the Go button to simulate this project and generate a waveform similar to that shown in Figure 5-12.

2. Use the Zoom In Button and the horizontal scroll bar to change the display range as shown.

Working with Waveforms and Cursors

The PeakSIM Waveform Display has a variety of features for examining waveform results, saving and printing waveforms and measuring times between events.

The Waveform Display has a dynamic cursor that allows you to quickly pan across a dense (zoomed out) waveform and observe values:

3. Move the mouse pointer over the waveform display and observe the changing values displayed in the signal display area (the small window to the left of the waveform).

The Waveform Display also includes selectable cursors that can be used to accurately measure the distances between events, and to view the timing relationships between events on different signals.

To add a cursor to the Waveform Display:

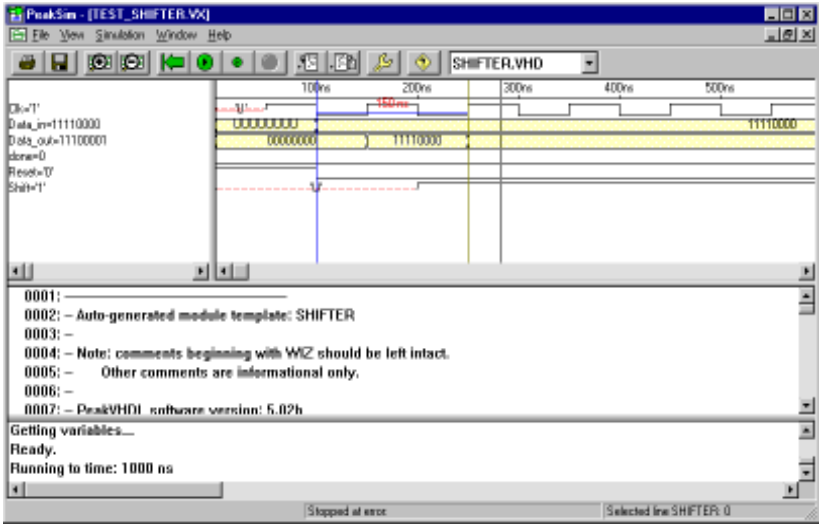
4. Click the mouse button within the Waveform Display window.

A new cursor is displayed each time you click the mouse button. When you add multiple cursors, PeakSIM adds a measurement line and value allowing you to quickly determine the time between two or more events (Figure 5-13).

To delete the cursors you have placed:

5. Select the **Remove All Cursors** item from the View menu.

Figure 5-13: *Click the mouse button over the waveform display to add measurement cursors. Use the Remove All Cursors from the View menu to remove the cursors.*



Summary

This introductory tutorial has covered only the basics of VHDL simulation using the PeakSIM application. The simulator supports additional features such as text I/O that allow your VHDL design to interact with the de-simulation environment, and includes includes source-level bugging features, which are described in the next chapter.

Chapter 6: Using the Debug Window

PeakVHDL™ Professional Edition includes a powerful feature called *source-level debugging* that allows you to observe how your VHDL design is being executed during simulation. Using this feature, you will be able to step through your VHDL code, set breakpoints, and more easily find and fix problems in your VHDL design description.

Understanding Source-Level Debugging

The source-level debug window allows you to follow the execution of your VHDL design at the level of VHDL source file statements. This is useful for debugging complex sequential statements, determining the order in which statements are processed, and understanding the impact of scheduling, delta cycles and other complex aspects of model execution.

To allow source-level debugging to be performed, the PeakVHDL linker must insert certain precompiled code statements into your simulation executable. These statements are not visible to you, except that you may notice your compiled VHDL projects require more disk space after linking with source-level debugging enabled.

During simulation of your design, PeakVHDL keeps track of which VHDL source file lines are related to the currently executing compiled and linked code, and displays the appropriate VHDL source file in a source file display window. In addition, PeakVHDL maintains a list of breakpoints that you have requested and stops the simulation whenever one of these break points is encountered. It then waits for you to either continue the simulation using the Go or Step Time buttons, or single-step through your code using the Step Over or Step Into buttons.

Whenever the simulator stops at a break point or is stepped to a new line in the VHDL source file, the waveform window is updated to display the current values of all selected signals. This feature allows you to observe the order in which signals and variables are updated in your design, and allows you to (for example) determine when you have incorrectly specified a signal or variable assignment.

A Sample Project

To give you a better understanding of source-level debugging, we'll present a sample project and show how it is compiled and run. You can follow along with this example by first opening the PeakVHDL standard example **getpizza,** which can be found in the examples directory of your PeakVHDL installation area.

The getpizza example project is intended as an exercise in writing test benches, and is also useful for demonstrating the concepts of source level debugging. At the center of getpizza is a driving game that was inspired by the "ChipTrip" example first described by Altera Corporation using their AHDL PLD language. In our version of the design (which is described in more detail in *VHDL Made Easy*, published in 1996 by Prentice Hall), the objective is to create a sequence of test inputs that will cause an imaginary work-weary engineer to proceed from his office to the beach, as quickly as possible,

without getting a speeding ticket. To make the trip more interesting, our hero must stop and pick up a pizza on the way. The map of Figure 6-1 illustrates the possible routes that can be taken.

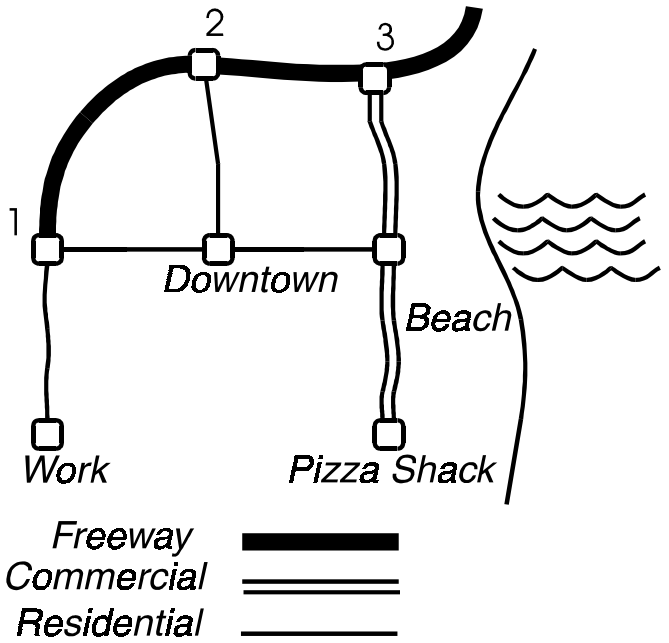


Figure 6-1: *The sample project we'll use is an exercise in test bench development: a driving game.*

This map shows three different types of roads: freeways, commercial streets, and residential roads. The car being driven has only two possible speeds, fast and slow. When the car is driven slowly, it advances from one point on the map (say, from **Ramp1** to **Ramp2**) in a given period of time. When driven fast, the car proceeds twice as far. There is no speed limit on the freeway, so the car can travel at full speed without fear of getting a ticket. On commercial streets, the car may exceed the speed limit just once and get away with it. On residential roads, any attempt to drive fast will result in a ticket.

In our simulation, and in the underlying design description, a fixed period of time is represented by a single clock cycle. Inputs for the speed and initial direction of travel are represented by signals **Speed** and **Dir**. The location of the car at any point is represented internally to the circuit by a state machine, but it is kept hidden at the top level of the design and in the test bench itself. The current status and success or failure of a trip are observed on the signals **DriveTime**, **Tickets**, and **Party**, which tell the player how long the drive has taken, how many traffic tickets have accrued, and whether he or she has yet arrived at the beach with the pizza. (The VHDL source files and PeakVHDL project file for the entire design can be found in your **examples\vhdl93\getpizza** installation directory.)

The test bench that we have written for this design reads symbolic test commands from a file, allowing the game to be easily tested and various driving scenarios to be described without having to recompile the design each time.

Loading the Sample Project

To load the sample project,

1. Invoke PeakVHDL and select the Open Project button or select the **Open Project** item from the **File** menu.
2. Navigate to the **examples\vhdl93\getpizza** directory and choose the **getpizza.acc** project file.

After you have opened the project, you will see that there are four VHDL modules listed in the Hierarchy Browser. (You can invoke the text editor to examine these source files if you wish. To invoke the text editor, double-click on any entry in the Hierarchy Browser as shown in Figure 6-2.) The **testpizza** module describes the test bench for this project, so select that module by clicking once on the **MODULE TESTPIZZA** entry in the Hierarchy Browser.

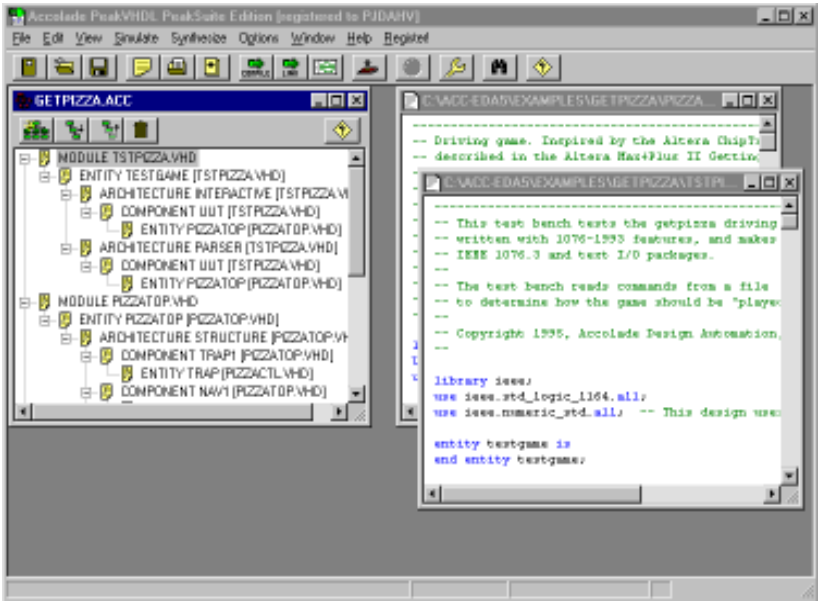


***Figure 6-2**: Open the GETPIZZA project to begin. The project includes four VHDL source files.*

Setting Project Options

Before processing this project for simulation, we'll need to set certain project options to enable source-level debugging. To set these options,

1. Click the Options button, the select the Link tab (or select **Link Options** from the **Options** menu).

2. Set the options as shown in Figure 6-3.

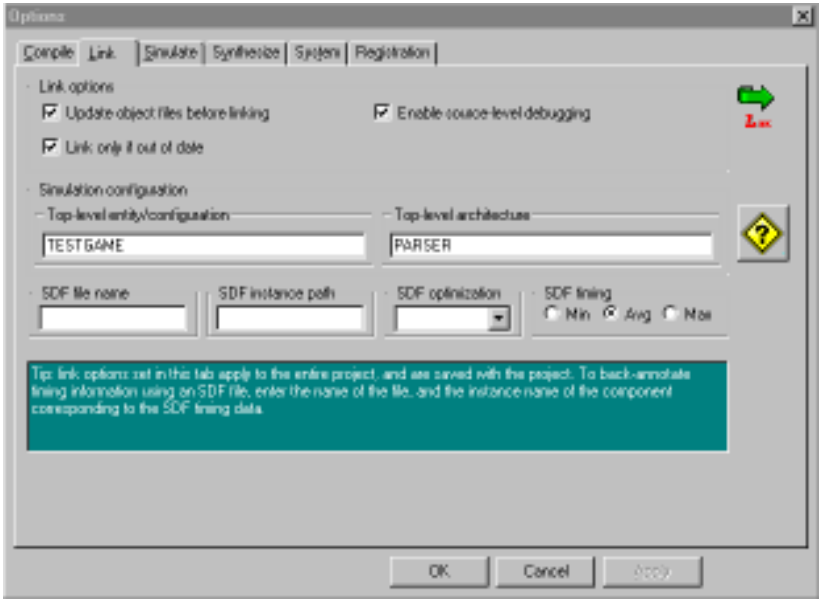


Figure 6-3*: To enable source-level debugging, be sure the* ***Enable source-level debugging*** *check box is selected in the Link Options dialog.*

The important Link option being set for this example is the **Enable source level debugging** option. This option causes debugging code to be added to the compiled and linked simulation executable. Also note that the 1076-1993 option is set. This is required because the **getpizza** example has been written using features of the IEEE 1076-1993 language specification.

Loading the Simulation

Our sample project is now ready for simulation. To load this project for simulation,

1. Select (highlight) the **testpizza** module by clicking on the **MODULE TESTPIZZA** entry in the Hierarchy Browser.

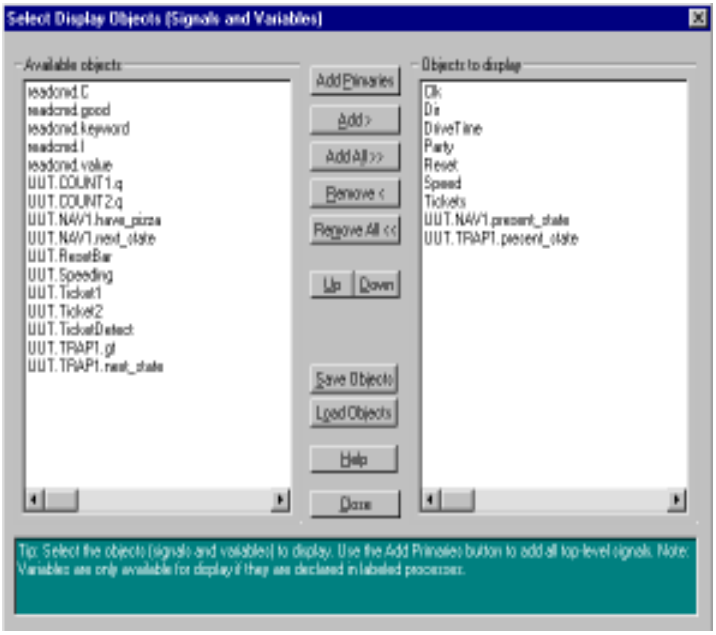


Figure 6-4: *Use the Select Display Objects dialog to select signals for display. You can select the two state register signals as shown.*

2. Click the Load button, or select the **Load Selected** item from the **Simulate** menu.

Before loading the project, PeakVHDL will compile all VHDL modules and link them to create a simulation executable. After the simulation executable has been loaded, PeakVHDL will display a Signal Selection dialog (Figure 6-4) allowing you to select signals for display.

To select signals,

3. Use the Available Signals window to select some or all of the signals in the design, or use the Add Primaries button to select all top-level signals in the design.

After you have selected signals for display, the waveform display and source-level debug windows appear as shown in Figure 6-5.

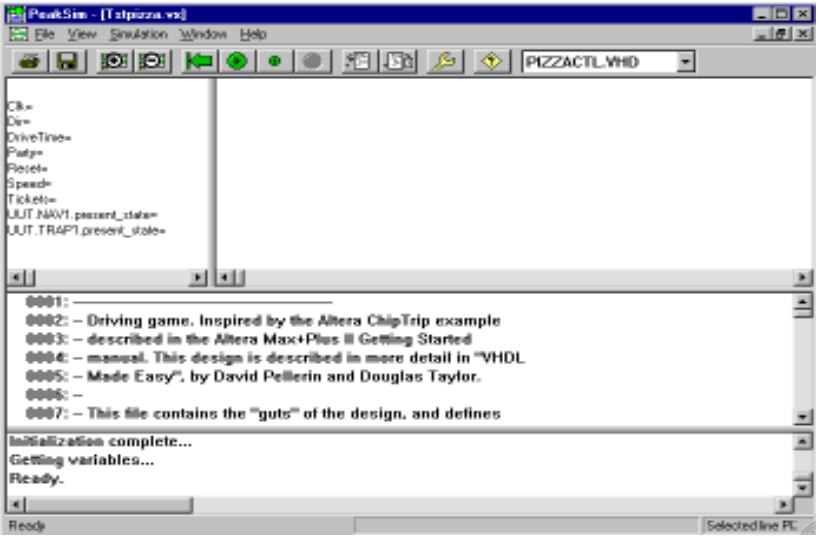


Figure 6-5: *The PeakSIM application includes a signal display window (upper left), a waveform display window (upper right), a source code window (center) and a transcript window (bottom).*

Setting a Break Point

You can set or remove breakpoints before starting the simulation, or at any time the simulation is stopped. Before starting a simulation run, set a break point in the **pizzactl** module.

To set the break point,

1. Use the module name drop-down list box to select the **pizzactl** module.
2. Scroll through the **pizzactl** module and find the source file line shown in Figure 6-6.
3. Set a break point at the indicated line number by double-clicking or by using the Toggle Breakpoint menu item from the Simulation menu.

Running Simulation

Now you can start the simulation and let it run to the selected break point. To start the simulation,

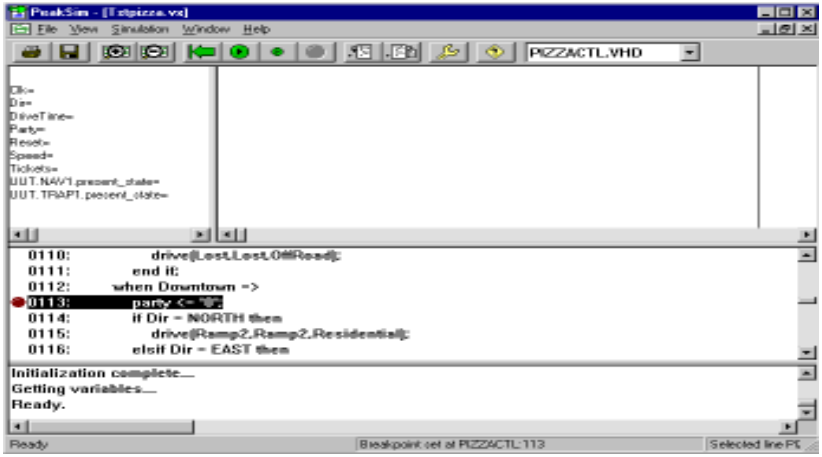


Figure 6-6: *Scroll through the* ***pizzactl*** *module to find the source file line shown.*

1. Click the Go button.

The simulation will execute until it encounters the breakpoint that you have selected, or until it reaches the specified simulation end time if no breakpoint is encountered. It will then stop execution and display the current module and source file line in the source file display window. It will also update the waveform display so you can see the current values of all signals and variables being displayed.

At this point you could single-step the design to view exactly how the state machine represented by this section of code operates, or click the Go button to continue to the next breakpoint (or to the specified simulation end time, if there are no more breakpoints set).

Try using the single stepping features. To single-step the simulation,

2. Click the Step Over button repeatedly until the current line pointer (the green arrow icon) in at the position shown in Figure 6-7.

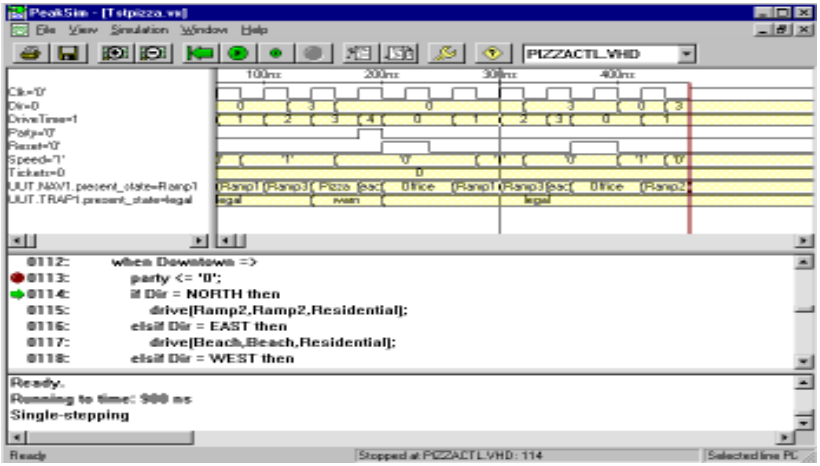


Figure 6-7: *The Step Over button allows you to trace the execution of your VHDL source code line-by-line.*

The simulation is now stopped at a source file line that includes a call to a procedure named **Drive**. This procedure is defined elsewhere in the module (at the top of the architecture).

You can use the Step Over button at this point to continue to the next line in the file, or use the Step Into button to cause the simulation to enter the **Drive** procedure and stop at the first line in that procedure.

Try setting other breakpoints in the source files and contibue the simulation (using the Go button) to get a feel for source-level debugging.

Summary

This tutorial has shown how source-level debugging can be used to examine the execution of a VHDL design with more precision than is possible using only waveforms. When you combine source level-debugging with PeakVHDL's waveform display and export features, and the text I/O features of VHDL, you have a powerful set of debugging tools at your disposal.

Chapter 7: A First Look at VHDL

This chapter will introduce you to VHDL and show you how the language can be used to describe circuits for simulation and synthesis. This chapter is not intended as a comprehensive VHDL reference, but will give you enough information to quickly get started using VHDL. Along the way, we will suggest coding styles that are appropriate using a wide variety of available synthesis and simulation tools, including PeakVHDL.

During this introduction to VHDL, you will see some of the many advantages of using VHDL for synthesis and simulation.

What Is VHDL?

VHDL is a programming language that has been designed and optimized for describing the behavior of hardware digital circuits and systems. As such, VHDL combines features of a simulation modeling language, a design entry language, a test language, and a netlist language.

As a simulation modeling language, VHDL includes many features appropriate for describing the behavior of electronic components ranging from simple logic gates to complete microprocessors and custom chips. Features of VHDL allow electrical aspects of circuit behavior (such as rise and fall times of signals, delays through gates, and functional operation) to be precisely described. The resulting VHDL simulation models can then be used as building blocks in larger circuits (using schematics, block diagrams or system-level VHDL descriptions) for the purpose of simulation.

Just as high-level programming languages allow complex design concepts to be expressed as computer programs, VHDL allows the behavior of complex electronic circuits to be captured into a design system for automatic circuit synthesis or for system simulation. This process is called design entry, and is the first step taken when a circuit concept is to be realized using computer-aided design tools.

Design entry using VHDL is very much like software design using a software programming language. Like Pascal, C and C++, VHDL includes features useful for structured design techniques, and offers a rich set of control and data representation features. Unlike these other programming languages, VHDL provides features allowing concurrent events to be described. This is important because the hardware being described using VHDL is inherently concurrent in its operation. Users of PLD programming languages such as PALASM, ABEL, CUPL and others will find the concurrent features of VHDL quite familiar. Those who have only programmed using software programming languages will have some new concepts to grasp.

One area where hardware design differs from software design is in the area of testing. One of the most important (and underutilized) aspects of VHDL is its use as a way to capture the performance specification for a circuit, in the form of what is commonly referred to as a test bench. Test benches are VHDL descriptions of circuit stimulus and corresponding expected

outputs that verify the behavior of a circuit over time. Test benches should be an integral part of any VHDL project and should be created in parallel with other descriptions of the circuit.

And, while VHDL is a powerful language with which to enter new designs at a high level, it is also useful as a low-level form of communication between different tools in a computer-based design environment. VHDL's structural language features allow it to be effectively used as a netlist language, replacing (or augmenting) other netlist languages such as EDIF.

VHDL: A Standard Language

One of the most compelling reasons for you to become experienced with and knowledgeable in VHDL is its adoption as a standard in the electronic design community. Using a standard language such as VHDL virtually guarantees that you will not have to throw away and recapture design concepts simply because the design entry method you have chosen is not supported in a newer generation of design tools. Using a standard language also means that you are more likely to be able to take advantage of the most up-to-date design tools and that you will have access to a knowledge base of thousands of other engineers, many of whom are solving problems similar to your own.

A Brief History Of VHDL

VHDL (which stands for VHSIC hardware description language) was developed in the early 1980s as a spin-off of a high-speed integrated circuit research project funded by the U.S. Department of Defense. During the VHSIC program, researchers were confronted with the daunting task of describing circuits of enormous scale (for their time) and of managing very large circuit design problems that involved multiple

teams of engineers. With only gate-level design tools available, it soon became clear that better, more structured design methods and tools would be needed.

IEEE Standard 1076

To meet this challenge, a team of engineers from three companies — IBM, Texas Instruments and Intermetrics — were contracted by the Department of Defense to complete the specification and implementation of a new, language-based design description method. The first publicly available version of VHDL, version 7.2, was released in 1985. In 1986, the Institute of Electrical and Electronics Engineers, Inc. (IEEE) was presented with a proposal to standardize the language, which it did in 1987 after substantial enhancements and modifications were made by a team of commercial, government and academic representatives. The resulting standard, IEEE 1076-1987, is the basis for virtually every VHDL simulation and synthesis product sold today. An enhanced and updated version of the language, IEEE 1076-1993, was released in 1994, and VHDL tool vendors have been responding by adding these new language features to their products.

IEEE Standard 1164

Although IEEE Standard 1076 defines the complete VHDL language, there are aspects of the language that make it difficult to write completely portable design descriptions (descriptions that can be simulated identically using different vendors' tools). The problem stems from the fact that VHDL supports many abstract data types, but it does not address the simple problem of characterizing different signal strengths or commonly used simulation conditions such as unknowns and high-impedance.

Soon after IEEE 1076-1987 was adopted, simulator companies began enhancing VHDL with new signal types (typically through the use of syntactically legal, but nonstandard enumerated types) to allow their customers to accurately simulate complex electronic circuits. This caused problems because

design descriptions entered using one simulator were often incompatible with other simulation environments. VHDL was quickly becoming a nonstandard.

To get around the problem of nonstandard data types, another standard was created by an IEEE commitee. This standard, numbered 1164, defines a standard package (a VHDL feature that allows commonly used declarations to be collected into an external library) containing definitions for a standard nine-valued data type. This standard data type is called **std_logic**, and the IEEE 1164 package is often referred to as the standard logic package, or MVL9 (for multi-valued logic, nine values).

The IEEE 1076-1987 and IEEE 1164 standards together form the complete VHDL standard in widest use today. (IEEE 1076-1993 is slowly working its way into the VHDL mainstream, but it does not add significant new features for synthesis users.)

IEEE Standard 1076.3 (Numeric Standard)

Standard 1076.3 (often called the Numeric Standard or Synthesis Standard) defines standard packages and interpretations for VHDL data types as they relate to actual hardware. This standard is intended to replace the many custom (nonstandard) packages that vendors of synthesis tools have created and distributed with their products.

IEEE Standard 1076.3 does for synthesis users what IEEE 1164 did for simulation users: increase the power of Standard 1076, while at the same time ensuring compatibility between different vendors tools. The 1076.3 standard includes, among other things:

- A documented hardware interpretation of values belonging to the **bit** and **boolean** types defined by IEEE Standard 1076, as well as interpretations of the **std_ulogic** type defined by IEEE Standard 1164.

- A function that provides “don’t care” or “wild card” testing of values based on the **std_ulogic** type. This is of particular use for synthesis, since it is often helpful to express logic in terms of “don’t care” values.
- Definitions for standard **signed** and **unsigned** arithmetic data types, along with arithmetic, shift, and type conversion operations for those types.

IEEE Standard 1076.4 (VITAL)

The annotation of timing information to a simulation model is an important aspect of accurate digital simulation. The VHDL 1076 standard describes a variety of language features that can be used for timing annotation; however, it does not describe a standard method for expressing timing data outside of the timing model itself.

The ability to separate the behavioral description of a simulation model from the timing specifications is important for many reasons. One of the major strengths of Verilog HDL (VHDL’s closest rival) is the fact that Verilog HDL includes a feature specifically intended for timing annotation. This feature, the *Standard Delay Format*, or *SDF*, allows timing data to be expressed in a tabular form and included into the Verilog timing model at the time of simulation.

The IEEE 1076.4 standard, published by the IEEE in late 1995, adds this capability to VHDL as a standard package. A primary impetus behind this standard effort (which was dubbed *VITAL*, for *VHDL Initiative Toward ASIC Libraries*) was to make it easier for ASIC vendors and others to generate timing models applicable to both VHDL and Verilog HDL. For this reason, the underlying data formats of IEEE 1076.4 and Verilog’s SDF are quite similar.

Learning VHDL

This chapter presents several sample circuits and shows how they can be described for synthesis and testing. These small examples are not intended to represent real applications, but will help you to understand the relationships between various types of VHDL statements and the actual hardware being described.

In addition to the quick introduction to VHDL presented in this chapter, there are some very important concepts that will be introduced. Perhaps the most important concepts to understand in VHDL, are those of *concurrency* and *hierarchy*. Since these concepts are so important (and may be new to you), we will introduce both concurrency and hierarchy in these initial examples.

Before visiting these more complex topics. however, we will first present a very simple example so you can see what constitutes the minimum VHDL source file.

We'll start this chapter off by looking at a very simple combinational circuit: an 8-bit comparator.

Our comparator will accept two 8-bit inputs, compare them, and produce a 1-bit result (either 1, indicating a match, or 0, indicating a difference between the two input values). A comparator such as this is a combinational function constructed in circuitry from an arrangement of exclusive-OR gates or from some other lower-level structure depending on the capabilities of the target technology. (It is the job of logic synthesis to determine exactly what hardware representation is most appropriate for a given device.)

```
entity compare is
   port(A,B: in bit;
        EQ: out bit);
end compare;

architecture compare1 of compare is
begin
```

```
    EQ <= '1' when (A = B) else '0';

end compare1;
```

Note: *In this and other VHDL source files listed in this manual, VHDL keywords are highlighted in bold face type.*

Let's look more closely at this source file. Reading from the top, we see the following elements:

- An entity declaration that defines the inputs and outputs — the ports — of this circuit; and
- An architecture declaration that defines what the circuit actually does, using a single concurrent assignment.

Entities and Architectures

Every VHDL design description consists of at least one entity / architecture pair. (In VHDL jargon, this combination of an entity and its corresponding architecture is sometimes referred to as *a design entity*.) In a large design, you will typically write many entity / architecture pairs and connect them together to form a complete circuit.

An *entity declaration* describes the circuit as it appears from the "outside" - from the perspective of its input and output interfaces. If you are familiar with schematics, you might think of the entity declaration as being analogous to a block symbol on a schematic.

The second part of a minimal VHDL design description is the *architecture declaration*. Every entity in a VHDL design description must be bound with a corresponding architecture. The architecture describes the actual function — or contents — of the entity to which it is bound. Using the schematic as a

metaphor, you can think of the architecture as being roughly analogous to a lower-level schematic pointed to by the higher-level functional block symbol.

Entity Declaration

An entity declaration provides the complete interface for a circuit. Using the information provided in an entity declaration (the names, data types and direction of each port), you have all the information you need to connect that portion of a circuit into other, higher-level circuits, or to develop input stimulus (in the form of a test bench) for testing purposes. The actual operation of the circuit, however, is not included in the entity declaration.

Let's take a closer look at the entity declaration for this simple design description:

```
entity compare is
   port( A, B: in bit_vector(0 to 7);
         EQ: out bit);
end compare;
```

The entity declaration includes a name, **compare**, and a *port declaration* statement defining all the inputs and outputs of the entity. The port list includes definitions of three ports: **A**, **B**, and **EQ**. Each of these three ports is given a direction (either **in**, **out** or **inout**), and a type (in this case either **bit_vector(0 to 7)**, which specifies an 8-bit array, or **bit**, which represents a single-bit value).

There are many different data types available in VHDL. To simplify things in this introductory circuit, we're going to stick with the simplest data types in VHDL, which are **bit** and **bit_vector**.

Architecture Declaration

The second part of a minimal VHDL source file is the architecture declaration. Every entity declaration you write must be accompanied by at least one corresponding architecture.

Here's the architecture declaration for the comparator circuit:

```
architecture compare1 of compare is
begin

   EQ <= '1' when (A = B) else '0';

end compare1;
```

The architecture declaration begins with a unique name, **compare1**, followed by the name of the entity to which the architecture is bound, in this case **compare**. Within the architecture declaration (between the **begin** and **end** keywords) is found the actual functional description of our comparator. There are many ways to describe combinational logic functions in VHDL; the method used in this simple design description is a type of concurrent statement known as a *conditional assignment*. This assignment specifies that the value of the output (**EQ**) will be assigned a value of '1' when **A** and **B** are equal, and a value of '0' when they differ.

This single concurrent assignment demonstrates the simplest form of a VHDL architecture. As you will see, there are many different types of concurrent statements available in VHDL, allowing you to describe very complex architectures. Hierarchy and subprogram features of the language allow you to include lower-level components, subroutines and functions in your architectures, and a powerful statement known as a *process* allows you to describe complex sequential logic as well.

Data Types

Like a high-level software programming language, VHDL allows data to be represented in terms of high-level data types. These data types can represent individual wires in a circuit, or they can represent collections of wires using a concept called an *array*.

The preceding description of the comparator circuit used the data types **bit** and **bit_vector** for its inputs and outputs. The **bit** data type has only two possible values: '1' or '0'. (A **bit_vector** is simply an array of **bits.**) Every data type in VHDL has a defined set of values, and a defined set of valid operations. Type checking is strict, so it is not possible, for example, to directly assign the value of an **integer** data type to a **bit_vector** data type. (There are ways to get around this restriction, using what are called *type conversion functions*. These are not discussed in this manual, but examples of their use are provided in Appendix C, *Examples Gallery*.)

The following chart summarizes the fundamental data types available in VHDL.

Data Type	*Values*	*Example*
Bit	'1', '0'	Q <= '1';
Bit_vector	(array of bits)	DataOut <= "00010101";
Boolean	True, False	EQ <= True;
Integer	-2, -1, 0, 1, 2, 3, 4, etc.	Count <= Count + 2;
Real	1.0, -1.0E5, etc.	V1 = V2 / 5.3
Physical	1 ua, 7 ns, 100 ps, etc.	Q <= '1' after 6 ns;
Record	(various)	Tvec := (Clk, Inp, Result);
Character	'a', 'b', '2, '$', etc.	CharData <= 'X';
String	(Array of characters)	Msg <= "MEM: " & Addr

Design Units

One concept unique to VHDL (when compared to software programming languages and to its main rival, Verilog HDL) is the concept of a design unit. Design units in VHDL (which may also be referred to as *library units*) are segments of VHDL code that can be compiled separately and stored in a library. You have been introduced to two design units already: the entity and the architecture. There are actually five types of design units in VHDL; entities, architectures, packages, package bodies, and configurations.

Figure 7-1 illustrates the relationship between these five design units:

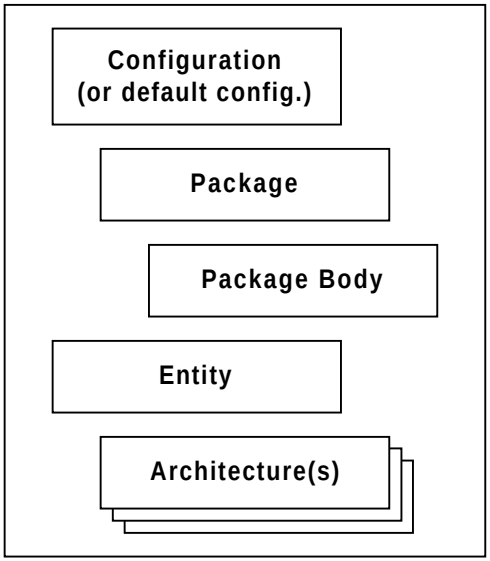


Figure 7-1: *VHDL includes five types of design units: configurations, packages, package bodies, entities and architectures.*

Entities

A VHDL *entity* is a statement (indicated by the **entity** keyword) that defines the external specification of a circuit or subcircuit. The minimum VHDL design description must include at least one entity and one corresponding architecture.

When you write an entity declaration, you must provide a unique name for that entity and a port list defining the input and output ports of the circuit. Each port in the port list must be given a name, direction (or *mode*, in VHDL jargon) and a type. Optionally, you may also include a special type of parameter list (called a *generic list*) that allows you to pass additional information into an entity.

Architectures

A VHDL architecture declaration is a statement (beginning with the **architecture** keyword) that describes the underlying function and / or structure of a circuit. Each architecture in your design must be associated (or *bound*) by name with one entity in the design.

VHDL allows you to create more than one alternate architecture for each entity. This feature is particularly useful for simulation and for project team environments in which the design of the system interfaces (expressed as entities) is done by a different engineer than the lower-level architectural description of each component circuit.

An architecture declaration consists of zero or more declarations (of items such as intermediate signals, components that will be referenced in the architecture, local functions and procedures, and constants) followed by a **begin** statement, a series of concurrent statements, and an **end** statement.

Packages and Package Bodies

A VHDL package declaration is identified by the **package** keyword, and is used to collect commonly-used declarations for use globally among different design units. You can think of

a package as a common storage area, one used to store such things as type declarations, constants, and global subprograms. Items defined within a package can be made visible to any other design unit in the complete VHDL design, and they can be compiled into libraries for later re-use.

A package can consist of two basic parts: a package declaration and an optional package body. Package declarations can contain the following types of statements:

- Type and subtype declarations
- Constant declarations
- Global signal declarations
- Function and procedure declarations
- Attribute specifications
- File declarations
- Component declarations
- Alias declarations
- Disconnect specifications
- Use clauses

Items appearing within a package declaration can be made visible to other design units through the use of a **use** statement, as we will see.

If the package contains declarations of subprograms (functions or procedures) or defines one or more deferred constants (constants whose value is not immediately given), then a *package body* is required in addition to the package declaration. A package body (which is specified using the **package body** keyword combination) must have the same name as its corresponding package declaration, but it can be located anywhere in the design (it does not have to be located immediately after the package declaration).

The relationship between a package and package body is somewhat akin to the relationship between an entity and its corresponding architecture. (There may be only one package body written for each package declaration, however.) While the package declaration provides the information needed to use the items defined within it (the parameter list for a global procedure, or the name of a defined type or subtype), the actual behavior of such things as procedures and functions must be specified within package bodies.

Examples of global procedures and functions can be found in Appendix C, *Examples Gallery*.

Configurations

The final type of design unit available in VHDL is called a configuration declaration. You can think of a configuration declaration as being roughly analogous to a parts list for your design. A configuration declaration (identified with the **configuration** keyword) specifies which architectures are to be bound to which entities, and it allows you to change how components are connected in your design description at the time of simulation or synthesis.

Configuration declarations are always optional, no matter how complex a design description you create. In the absence of a configuration declaration, the VHDL standard specifies a set of rules that provide you with a default configuration. For example, in the case where you have provided more than one architecture for an entity, the last architecture compiled will take precedence and will be bound to the entity.

Levels of Abstraction (Styles)

VHDL supports many possible styles of design description. These styles differ primarily in how closely they relate to the underlying hardware. When we speak of the different styles of

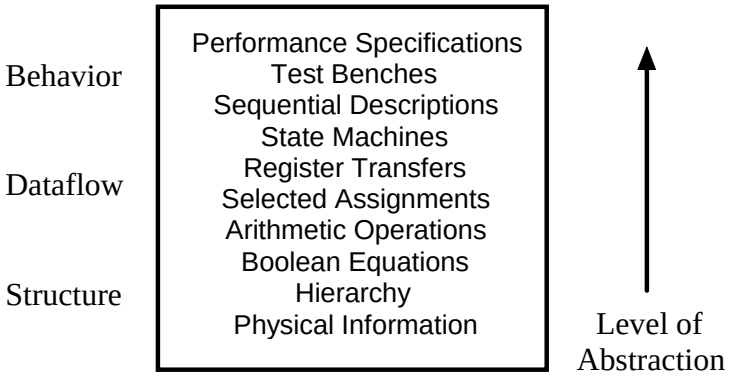


Figure 7-2: *VHDL allows you to describe a system at many levels of abstraction.*

VHDL, we are really talking about the differing levels of abstraction possible using the language — behavior, dataflow, and structure — as shown in Figure 7-2:

This figure maps the various points in a top-down design process to the three general levels of abstraction. Starting at the top, suppose the performance specifications for a given project are: “the compressed data coming out of the DSP chip needs to be analyzed and stored within 70 nanoseconds of the Strobe signal being asserted...” This human language specification must be refined into a description that can actually be simulated. A test bench written in combination with a sequential description is one such expression of the design. These are all points in the **behavior** level of abstraction.

After this initial simulation, the design must be further refined until the description is something a VHDL synthesis tool can digest. That is the **dataflow** level of abstraction.

The **structure** level of abstraction occurs when smaller segments of circuitry are being connected together to form a larger circuit. The structure level is what we commonly think of as a circuit netlist, or perhaps a higher-level block diagram.

Behavior

The highest level of abstraction supported in VHDL is called the *behavior* level of abstraction. When creating a behavioral description of a circuit, you will describe your circuit in terms of its operation *over time*. The concept of time is the critical distinction between behavioral descriptions of circuits and lower-level descriptions (specifically descriptions created at the dataflow level of abstraction).

In a behavioral description, the concept of time may be expressed precisely, with actual delays between related events (such as the propagation delays within gates and on wires), or it may simply be an ordering of operations that are expressed sequentially (such as in a functional description of a flip-flop). When you are writing VHDL for input to synthesis tools, you may use behavioral statements in VHDL to imply that there are registers in your circuit. It is unlikely, however, that your synthesis tool will be capable of creating precisely the same behavior in actual circuitry as you have defined in the language. (Synthesis tools today ignore detailed timing specifications, leaving the actual timing results at the mercy of the target device technology.)

If you are familiar with event-driven software programming, writing behavior-level VHDL will not seem like anything new. Just like a programming language, you will be writing one or more small programs that operate sequentially and communicate with one another through their interfaces. The only difference between behavior-level VHDL and a software programming language is the underlying execution platform: in the case of software, it is some operating system running on a CPU; in the case of VHDL, it is the simulator.

Dataflow

The *dataflow* level of abstraction is familiar to most digital circuit designers. In the dataflow level of abstraction, you describe your circuit in terms of how data moves through the system. At the heart of most digital systems today are regis-

ters, so in the dataflow level of abstraction you describe how information is passed between registers in the circuit. You will probably describe the combinational logic portion of your circuit at a relatively high level (and let a synthesis tool figure out the detailed implementation in logic gates), but you will likely be quite specific about the placement and operation of registers in the complete circuit.

Structure

The third level of abstraction, *structure*, is used to describe a circuit in terms of its components. Structure can be used to create a very low-level description of a circuit (such as a transistor-level description) or a very high-level description (such as a block diagram).

In a gate-level description of a circuit, for example, components such as basic logic gates and flip-flops might be connected in some logical structure to create the circuit. This is what is often called a *netlist*. For a higher-level circuit — one in which the components being connected are larger functional blocks — structure might simply be used to segment the design description into manageable parts.

Structure-level VHDL features such as components and configurations are very useful for managing complexity. The use of components can dramatically improve your ability to reuse elements of your designs, and they can make it possible to work using a *top-down* design approach.

Sample Circuit

To help demonstrate some of the important concepts we have covered in the first of this chapter, we will present a very simple circuit and show how the function of this circuit can be described in VHDL. The design descriptions we will show are intended for synthesis and therefore do not include timing specifications or other information not directly applicable to today's synthesis tools.

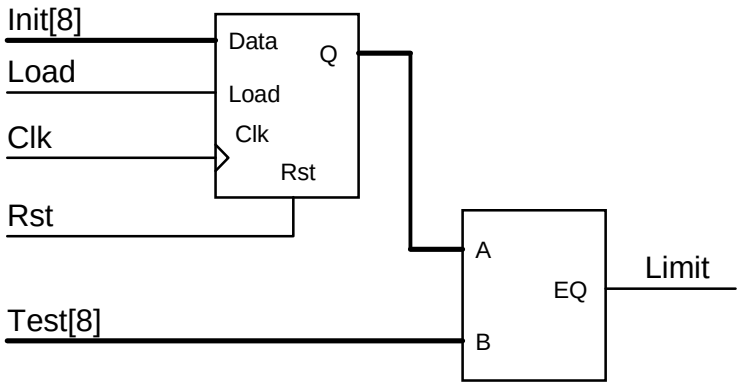


Figure 7-3: *The sample circuit includes an 8-bit shifter and a comparator.*

The circuit we will be looking at combines the comparator circuit presented earlier with a simple 8-bit loadable shift register. The shift register will allow us to examine in detail how behavior-level VHDL can be written for synthesis.

The two subcircuits (the shifter and comparator) will be connected using VHDL's hierarchy features and will demonstrate the third level of abstraction: *structure*. The complete circuit is shown in Figure 7-3.

This diagram has been intentionally drawn to look like a hierarchical schematic with each of the lower-level circuits represented as blocks. In fact, many of the concepts we will cover during the development of this circuit are the same concepts familiar to users of schematic hierarchy. These concepts include the ideas of component instantiation, mapping of ports, and design partitioning.

In a more structured project environment, you would probably enter a circuit such as this by first defining the interface requirements of each block, then describing the overall design of the circuit as a collection of blocks connected together through hierarchy at the top level. Later, after the system interfaces had been designed, you would proceed down the hierarchy (using a top-down approach to design) and fill in the details of each subcircuit.

In this example, however, we will begin by describing each of the lower-level blocks first and then connect them to form the complete circuit.

Comparator (Dataflow)

The comparator portion of the design will be identical to the simple 8-bit comparator we have already seen. The only difference is that we will use the IEEE 1164 standard logic data types (**std_ulogic** and **std_ulogic_vector**) rather than the **bit** and **bit_vector** data types used previously. Using standard logic data types for all system interfaces is highly recommended, as it allows circuit elements from different sources to be easily combined. It also provides you the opportunity to perform more detailed and precise simulation than would otherwise be possible.

The updated comparator design, using the IEEE 1164 standard logic data types, is shown below:

```
--------------------------------------
-- Eight-bit comparator
--
library ieee;
use ieee.std_logic_1164.all;
entity compare is
   port (A, B: in std_ulogic_vector(0 to 7);
         EQ: out std_ulogic);
end compare;

architecture compare1 of compare is
begin
   EQ <= '1' when (A = B) else '0';
end compare1;
```

Let's take a closer look at this simple VHDL design description. Reading from the top of the source file, we see:

- a comment field, indicated by the leading double-dash symbol ("--"). VHDL allows comments to be embedded anywhere in your source file, provided they are prefaced by the two hyphen characters as shown. Comments in VHDL extend from the double hyphen symbol to the end of the current line. (There is no block comment facility in VHDL.)
- a **library** statement that causes the named library IEEE to be loaded into the current compile session. When you use VHDL libraries, it is recommended that you include your library statements once at the beginning of the source file, before any **use** clauses or other VHDL statements.
- a **use** clause, specifying which items from the IEEE library are to be made visible for the subsequent design unit (the entity and its corresponding architecture). The general form of a **use** statement includes three fields delimited by a period: the library name (in this case **ieee**), a design unit within the library (normally a package, in this case named **std_logic_1164**), and the specific item within that design unit (or, as in this case, the special keyword **all**, which means everything) to be made visible.
- an entity declaration describing the interface to the comparator. Note that we have now specified **std_ulogic** and **std_ulogic_vector**, which are standard data types provided in the IEEE 1164 standard and in the associated IEEE library.
- an architecture declaration describing the actual function of the comparator circuit.

Conditional Signal Assignment

The function of the comparator is defined using a simple concurrent assignment to port **EQ**. The type of statement used in the assignment to **EQ** is called a *conditional signal assignment*. Conditional signal assignments make use of the **when-else** language feature and allow complex conditional logic to be described. The following description of a multiplexer circuit makes the use of the conditional signal assignment more clear:

```
architecture mux1 of mux is
begin

   Y <= A when (Sel = "00") else
        B when (Sel = "01") else
        C when (Sel = "10") else
        D when (Sel = "11");

end mux1;
```

Selected Signal Assignment

This form of signal assignment can be used as an alternative to the conditional signal assignment. The selected signal assignment has the following general form (again, using a multiplexer as an example):

```
architecture mux2 of mux is
begin

   with Sel select
      Y <= A when "00",
           B when "01",
           C when "10",
           D when "11";

end mux2;
```

Choosing between a conditional or selected signal assignment for circuits such as this is largely a matter of taste. For most designs, there is no difference in the results obtained with either type of assignment statement.

Barrel Shifter (Entity)

The second and most complex part of this design is the barrel shifter circuit. This circuit (diagrammed below) accepts 8-bit input data, loads this data into a register and, when the **load** input signal is low, rotates this data by one bit with each rising edge clock signal. The circuit is provided with an asynchronous reset, and the data stored in the register is accessible via the output signal **Q**.

They are many ways to describe a circuit such as this in VHDL. If you are going to use synthesis tools to process the design description into an actual device technology, however, you must restrict yourself to well established synthesis conventions when entering the circuit. We will examine two of these conventions when entering this design.

Using a Process

The first design description that we will look at for this shifter is a description that uses a VHDL **process** statement to describe the behavior of the entire circuit over time. This is the behavioral level of abstraction. It represents the highest level of abstraction practical (and synthesizable) for registered circuits such as this one. The VHDL source code for the barrel shifter is shown below:

```
-----------------------------------------
-- Eight-bit  barrel shifter
--
library ieee;
use ieee.std_logic_1164.all;
```

```
entity rotate is
   port( Clk, Rst, Load: in std_ulogic;
         Data: in std_ulogic_vector(0 to 7);
         Q: out std_ulogic_vector(0 to 7));
end rotate;

architecture rotate1 of rotate is
begin
   reg: process(Rst,Clk)
      variable Qreg: std_ulogic_vector(0 to 7);
   begin
      if Rst = '1' then   -- Async reset
         Qreg := "00000000";
      elsif (Clk = '1' and Clk'event) then
         if (Load = '1') then
            Qreg := Data;
         else
            Qreg := Qreg(1 to 7) & Qreg(0);
         end if;
      end if;
      Q <= Qreg;
   end process;
end rotate1;
```

Let's look closely at this source file. Reading from the top, we see:

- a comment field, as described previously.
- **library** and **use** statements, allowing us to use the IEEE 1164 standard logic data types.
- an entity declaration defining the interface to the circuit. Note that the direction (mode) of **Q** is written as **out**, indicating that it will not be used directly as the lower-level storage object (**Q** will not be fed back directly.).
- an architecture declaration, consisting of a single **process** statement that defines the operation of the shifter over time in response to events appearing on the clock (**Clk**) and asynchronous reset (**Rst**).

Process Statement

The **process** statement in VHDL is the primary means by which sequential operations (such as registered circuits) can be described. When describing registered circuits, the most common form of a **process** statement is:

```
architecture arch_name of ent_name is
begin
   process_name: process(sensitivity_list)
      local_declaration;
      local_declaration;
      . . .
   begin
      sequential statement;
      sequential statement;
      sequential statement;
      .
      .
      .
   end process;
end arch_name;
```

A process statement consists of the following items:

- An optional process name (an identifier followed by a colon character).
- The **process** keyword.
- An optional sensitivity list, indicating which signals result in the process "executing" when there is some event detected. (The sensitivity list is required if the process does not include one or more **wait** statements to suspend its execution at certain points. We will look at examples that do not use a sensitivity list later on in this chapter).
- An optional declarations section, allowing local objects and subprograms to be defined.

- A **begin** keyword.
- A sequence of statements to be executed when the program runs.
- an **end** statement.

The easiest way to think of a VHDL process such as this is to relate it to software, as a program that executes (in simulation) any time there is an event on one of its inputs (as specified in the sensitivity list). A process describes the sequential execution of statements that are dependent on one or more events occurring. A flip-flop is a perfect example of such a situation; it remains idle, not changing state, until there is a significant event (either a rising edge on the clock input or an asynchronous reset event) that causes it to operate and potentially change its state.

Although there is a definite order of operations within a process (from top to bottom), you can think of a process as executing in zero time. This means that (a) a process can be used to describe circuits functionally, without regard to their actual timing, and (b) multiple processes can be "executed" in parallel with little or no concern for which processes complete their operations first. (There are certain caveats to this behavior of VHDL processes. These caveats are described in detail in most VHDL textbooks.

Let's see how the process for our barrel shifter operates. For your reference, the process is shown below:

```
reg: process(Rst,Clk)
      variable Qreg: std_ulogic_vector(0 to 7);
   begin
      if Rst = '1' then   -- Async reset
         Qreg := "00000000";
      elsif (Clk = '1' and Clk'event) then
         if (Load = '1') then
            Qreg := Data;
         else
```

```
            Qreg := Qreg(1 to 7) & Qreg(0);
        end if;
    end if;
    Q <= Qreg;
end process;
```

As written, the process is dependent on (or *sensitive* to) the asynchronous inputs **Rst** and **Clk**. These are the only signals that can have events directly affecting the operation of the circuit; in the absence of any event on either of these signals, the circuit described by the process will simply hold its current value (that is, the process will remain suspended).

Now let's examine what happens when an event occurs on either one of these asynchronous inputs. First, consider what happens when the input **Rst** has an event in which it transitions to a high state (represented by the **std_ulogic** value of **'1'**). In this case, the process will begin execution and the first **if** statement will be evaluated. Because the event was a transition to **'1'**, the simulator will see that the specified condition (**Rst = '1'**) is true and the assignment of variable **Qreg** to the reset value of **"00000000"** will be performed. The remaining statements of the **if-then-elsif** expression (those that are dependent on the **elsif** condition) will be ignored. The final statement in the process, the assignment of output signal **Q** to the value of **Qreg**, is not subject to the **if-then-elsif** expression and is therefore placed on the process queue for execution. (signal assignments do not occur until the process actually suspends.) Finally, the process suspends, all signals that were assigned values in the process (in this case **Q**) are updated, and the process waits for another event on **Clk** or **Rst**.

What about the case in which there is an event on **Clk**? In this case, the process will again execute, and the **if-then-elsif** expressions will be evaluated in turn until a valid condition is encountered. If the **Rst** input continues to have a high value (a value of **'1'**), then the simulator will evaluate the first **if** test as true, and the reset condition will take priority. If, however, the **Rst** input is not a value of **'1'**, then the next expression (**Clk =**

'1' and Clk'event) will be evaluated. This expression is the most commonly-used convention for detecting clock edges in VHDL. To detect a rising edge clock, we write the expression **Clk = '1'** in the conditional expression, just as we did when detecting a reset condition. For this circuit, however, the expression **Clk = '1'** would not be specific enough, since the process may have begun execution as the result of an event on **Rst** that did not result in **Rst** transitioning to a **'1'**. (For example, a falling edge event on **Rst** — that is, a transition from 1 to 0 — would trigger the process but cause it to skip to the **elsif** statement even though there was no event on **Clk,** since the **Rst = 1** condition would evaluate as false.) To ensure that the event we are responding to is in fact an event on **Clk**, we use the built-in VHDL attribute **'event** to check if **Clk** was that signal triggering the process execution.

If the event that triggered the process execution was in fact a rising edge on **Clk**, then the simulator will go on to check the remaining **if-then** logic to determine which assignment statement is to be executed. If **Load** is determined to be **'1'**, then the first assignment statement is executed and the data is loaded from input **Data** to the registers. If **Load** is not **'1'**, then the data in the registers is shifted, as specified using the bit slice and concatenation operations available in the language.

Confusing? Perhaps; but if you simply use the style just presented as a template for describing registered logic and don't worry too much about the details of how it is executed during simulation, you won't have much trouble. Just keep in mind that every assignment to a variable or signal you make that is dependent on a **Clk = '1' and Clk'event** expression will result in at least one register when synthesized.

Signals and Variables

There are two fundamental types of objects used to carry data from place to place in a VHDL design description: signals and variables. In virtually all cases, you will want to use variables

to carry data between sequential operations (within processes, procedures and functions) and use signals to carry information between concurrent elements of your design (such as between two independent processes).

Examples of signals and variables, and differences between them, are shown in more detail in Appendix C, *Examples Gallery*. For now, it is useful to think of signals as wires (as in a schematic) and variables as temporary storage areas (similar to variables in a traditional software programming language).

In many cases, you can choose whether to use signals or variables to perform the same task. As a general rule, you should use variables whenever possible and use signals only when you must access data across different concurrent parts of your design.

Using a Procedure

As we have seen from the first version of the barrel shifter, describing registered logic using processes requires that you follow some established conventions (if you intend to synthesize the design) and to consider the behavior of the entire circuit. In the barrel shifter design description previously shown, the registers were implied by the placement and use of statements such as if **Clk = '1' and Clk'event**. Assignment statements subject to that clause resulted in D-type flip-flops being implied for the signals.

For smaller circuits, this mixing of combinational logic functions and registers is fine and not difficult to understand. For larger circuits, however, the complexity of the system being described can make such descriptions hard to manage, and the results of synthesis can often be confusing. For these circuits, it often makes more sense to retreat to a dataflow level of abstraction and to clearly define the boundaries between registered and combinational logic.

One easy way to do this is to remove the process from your design and replace it with a series of concurrent statements representing the combinational and registered portions of the circuit. The following VHDL design description uses this method to describe the same barrel shifter circuit previously described:

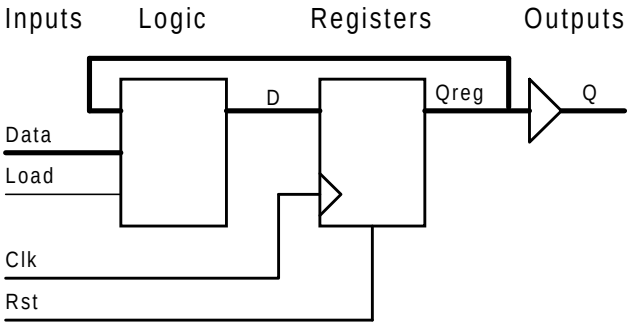
```
architecture rotate3 of rotate is
   signal D,Qreg: std_logic_vector(0 to 7);
begin

   D <= Data when (Load = '1') else
            Qreg(1 to 7) & Qreg(0);

   dff(Rst, Clk, D, Qreg);

   Q <= Qreg;

end rotate3;
```

In this version of the design description, the behavior of the D-type flip-flop has been placed in an external procedure, **dff()**, and intermediate signals have been introduced to more clearly describe the separation between the combinational and registered parts of the circuit. Figure 7-4 helps illustrate this separation:

In this example, the combinational logic of the counter has been written in the form of a single concurrent signal assignment, while the registered operation of the counter's output has been described using a call to a procedure named **dff**.

What does the **dff** procedure look like? The following is one possible procedure for a D-type flip-flop:

Figure 7-4: *Using a data flow level of abstraction can help simplify a design.Figure 10-1:*




```
procedure dff (signal Rst, Clk: in std_ulogic;
                signal D: in std_ulogic_vector(0 to 7);
                signal Q: out std_ulogic_vector(0 to 7)) is
begin
   if Rst = '1' then
      Q <= "00000000";
   elsif Clk = '1' and Clk'event then
      Q <= D;
   end if;
end dff;
```

Notice that this procedure has a striking resemblance to the process statement presented earlier. The same **if-then-elsif** structure used in the process is used to describe the behavior of the registers. Instead of a sensitivity list, however, the procedure has a parameter list describing the inputs and outputs of the procedure.

The parameters defined within a procedure or function definition are called its *formal parameters*. When the procedure or function is executed in simulation, the formal parameters are replaced by the values of the *actual parameters* specified when the procedure or function is used. If the actual parameters being passed into the procedure or function are signal objects, then the **signal** keyword can be used (as shown above) to ensure that all information about the signal object, including its value and all of its attributes, is passed into the procedure or function.

Structural VHDL

The structure level of abstraction is used to combine multiple components to form a larger circuit or system. As such, structure can be used to help manage a large and complex design, and structure can make it possible to reuse components of a system in other design projects.

Because structure only defines the interconnections between components, it cannot be used to completely describe the function of a circuit; at some level, all aspects of your circuit must be described using behavioral and/or dataflow levels of abstraction.

To demonstrate how the structure level of abstraction can be used to connect lower-level circuit elements into a larger circuit, we will connect the comparator and shift register circuits into a larger circuit as shown in Figure 7-5.

Notice that we have drawn this diagram in much the same way you might enter it into a schematic capture system. Structural VHDL has many similarities with schematic-based design, as we will see.

Design Hierarchy

When you write structural VHDL, you are in essence writing a textual description of a schematic *netlist* (a description of how the components on the schematic are connected by wires, or *nets*). In the world of schematic entry tools, such netlists are usually created for you automatically by the schematic editor. When writing VHDL, you enter the same sort of information by hand.

When you use components and wires (signals, in VHDL) to connect multiple circuit elements together, it is useful to think of your new, larger circuit in terms of a *hierarchy* of compo-

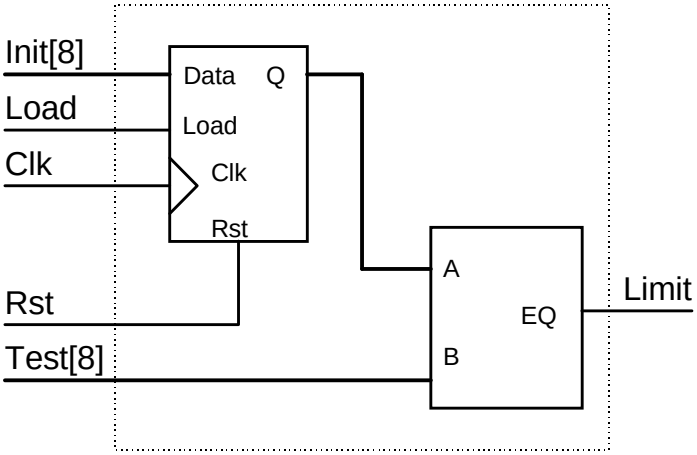


Figure 7-5: *The shifter and comparator are connected to form a larger system.*

nents. In this view, the top-level drawing (or top-level VHDL entity and architecture) can be seen as the highest level in a hierarchy tree, as shown in Figure 7-6.

```
library ieee;
use ieee.std_logic_1164.all;
entity rotcomp is port(Clk, Rst, Load: in std_ulogic;
                       Init: in std_ulogic_vector(0 to 7);
                       Test: in std_ulogic_vector(0 to 7);
                       Limit: out std_ulogic);
end rotcomp;

architecture structure of rotcomp is

   component compare
      port(A, B: in std_ulogic_vector(0 to 7); EQ: out std_ulogic);
   end component;
```

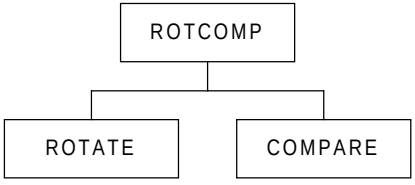


Figure 7-6: *The hierarchy of a design can be represented as a tree structure.*

```
  component rotate
    port(Clk, Rst, Load: in std_ulogic;
         Data: in std_ulogic_vector(0 to 7);
         Q: out std_ulogic_vector(0 to 7));
  end component;

  signal Q: std_ulogic_vector(0 to 7);

begin

  COMP1: compare port map (A=>Q, B=>Test, EQ=>Limit);
  ROT1: rotate port map (Clk=>Clk, Rst=>Rst, Load=>Load, Data=>Init,
                         Q=>Q);

end structure;
```

Test Benches

At this point, our sample circuit is complete and ready to be processed by synthesis tools. Before processing the design, however, we should take the time to verify that it actually does what it is intended to do. We should run a simulation.

Simulating a circuit such as this one requires that we provide more than just the design description itself. To verify the proper operation of the circuit over time in response to input stimulus, we will need to write a test bench.

The easiest way to understand the concept of a test bench is to think of it as a virtual tester circuit. This tester circuit, which you will describe in VHDL, applies stimulus to your design description and (optionally) verifies that the simulated circuit does what it is intended to do.

Figure 7-7 graphically illustrates the relationship between the test bench and your design description, which is called the *unit under test*, or *UUT*.

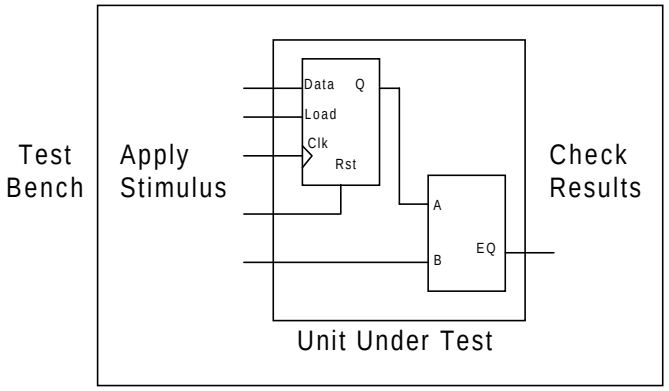


Figure 7-7: *The test bench forms a "virtual circuit" surrounding your design to be tested.*

To apply stimulus to your design, your test bench will probably be written using one or more sequential processes, and it will use a series of signal assignments and **wait** statements to describe the actual stimulus. You will probably use VHDL's looping features to simplify the description of repetitive stimulus (such as the system clock), and you may also use VHDL's file and record features to apply stimulus in the form of test vectors.

To check the results of simulation, you will probably make use of VHDL's assert feature, and you may also use the file features to write the simulation results to a disk file for later analysis.

For complex design descriptions, developing a comprehensive test bench can be a large-scale project in itself. In fact, it is not unusual for the test bench to be larger and more complex than the design description. For this reason, you should plan your project so that you have the time required to develop the function test in addition to developing the circuit being tested. You should also plan to create test benches that are re-usable, perhaps by developing a master test bench that reads test data from a file.

When you create a test bench for your design, you use the structural level of abstraction to connect your lower-level (previously top-level) design description to the other parts of the test bench.

Sample Test Bench

The following VHDL source statements describe a simple test bench for the shift and compare circuit. This test bench uses two processes that operate concurrently. One process (**clock**) describes a background clock with a 100 ns period, while the second process (**stimulus**) describes a sequence of inputs to be applied to the circuit over time. Note that this sample test bench does not include any checking of output values. More complex test benches that include output value checking are presented in Appendix C, *Examples Gallery*.

```
library ieee;
use ieee.std_logic_1164.all;

entity testbnch is                          -- No ports needed in a
end testbnch;                               -- testbench

architecture behavior of  testbnch is
   component rotcomp is                     -- Declares the lower-level
      port(Clk, Rst, Load: in std_ulogic;   -- component and its ports
           Init: in std_ulogic_vector(0 to 7);
           Test: in std_ulogic_vector(0 to 7);
           Limit: out std_ulogic);
   end component;
   signal Clk, Rst, Load: std_ulogic;       -- Introduces top-level signals
   signal Init: std_ulogic_vector(0 to 7);  -- to use when testing the
   signal Test: std_ulogic_vector(0 to 7);  -- lower-level circuit
   signal Limit: std_ulogic;
begin
```

```
   DUT: rotcomp port map                    -- Creates an instance of the
      (Clk, Rst, Load, Init, Test, Limit);  -- lower-level circuit (the
                                            -- design under test)
   clock: process
      variable clktmp: std_ulogic := '0';   -- This process sets up a
   begin                                    -- background clock of 100 ns
      clktmp := not clktmp;                 -- period.
      Clk <= clktmp;
      wait for 50 ns;
   end process;


   stimulus: process                        -- This process applies
   begin                                    -- stimulus to the design
      Rst <= '0';                           -- inputs, then waits for some
      Load <= '1';                          -- amount of time so we can
      Init <= "00001111";                   -- observe the results during
      Test <= "11110000";                   -- simulation.
      wait for 100 ns;
      Load <= '0';
      wait for 600 ns;
   end process;

end behavior;
```

Conclusion

In this chapter we have explored the most important concepts and features of VHDL. VHDL is a rich and powerful language, however, and there is much more to learn before you become a "master user". To continue your learning, it is strongly recommended that you acquire at least one textbook on VHDL, and also obtain a copy of the IEEE 1076 VHDL Language Reference Manual. There are also many good quality VHDL training courses and multimedia training products available. Contact Accolade Design Automation, or visit our Web page at www.acc-eda.com for more information.

You will also find it useful to study, copy and modify existing VHDL design examples. Appendix C of this manual includes listings and descriptions of a variety of sample designs, and additional examples are provided on your PeakVHDL installation CD-ROM.

Chapter 8: Using PeakLIB

PeakLIB is a utility that you can use to create PeakVHDL libraries and add or delete references to compiled VHDL modules (in the form of .AN files) from within existing library files.

PeakLIB Overview

PeakLIB is a DOS (command line) application that has been provided for PeakVHDL library creation and maintenance.

Why do you need PeakLIB? PeakVHDL library files (.LIB files) are created for you automatically when you specify a library name in the Compile Options dialog and compile a VHDL source file from within PeakVHDL. (If you have not specified a library file, the default library name of **WORK** is assumed.) This method of automatically creating libraries is fine for most projects, and is quite convenient because it allows you to associate library names with VHDL source files and have them automatically compiled into the correct library every time.

There are a few limitations inherent in creating libraries from within PeakVHDL, however:

1. PeakVHDL includes path (drive and directory) information in .LIB files that it generates. This makes it impossible to move existing projects to different drives or directories without recompiling the project, and makes library files essentially non-portable.
2. PeakVHDL does not provide any way to move existing .AN files from one library to another without recompiling the VHDL source file. In some cases (such as when you have purchased proprietary simulation models) you might not have access to the original VHDL source code.
3. PeakVHDL does not provide any way to remove a library entry from an existing library. For example, you may need to remove and replace entries in the IEEE library provided with PeakVHDL if you are using other VHDL tools that have specific requirements for IEEE library contents.

Examining the Contents of a Library File

PeakVHDL library files (.LIB files) are ASCII text files (with the exception of the first two characters in the file) and can be examing using any text editor, including the text editor provided in PeakVHDL. Each line of the library file includes a reference to a specific design unit located in the library and a corresponding reference to a compiled VHDL source file (an .AN file). The information in the library file is used by the PeakVHDL analyzer and elaborator (during the compile and link processes) to locate and use externally-referenced design units such as packages, components and lower-level entities.

Adding a .AN File to a Library

You can use PeakLIB to add an existing .AN file (compiled VHDL module) to a library file. If the library file name you specify does not already exist, it will be created.

To add an object file to a library or to create a new library, open a DOS window and type the command:

```
\ACC-EDA\PEAKLIB.EXE libname.LIB filename.AN
```

where *libname* is the name of the library (such as IEEE.LIB), and while *filename* is the name of an object file (such as NUM_STD.AN).

Deleting a .AN File Reference From a Library

To delete object files from a library, use the -D command, as in:

```
\ACC-EDA\PEAKLIB.EXE -D libname.LIB filename.AN
```

The reference to the .AN file will be removed from the library file.

Appendix A: Support Services

Learning More About VHDL

Although PeakVHDL has been designed to be as easy to use as possible, you will need to spend some time learning about VHDL before you are able to take full advantage of the language and the PeakVHDL product.

To gain knowledge and experience with VHDL, Accolade Design Automation strongly recommends that you purchase one or more VHDL textbooks, including the *IEEE Standard 1076 Language Reference Manual* available from the IEEE, phone (800) 678-4333.

A list of recommended VHDL textbooks and training resources can be found on the Accolade Web Site at the following URL: **http://www.acc-eda.com**. Textbooks and interactive computer-based training materials are also available direct from Accolade Design Automation. The Accolade Design Automation Web site also includes an on-line introduction to VHDL, as well as pointers to other VHDL-related sites and resources.

Obtaining Product Assistance

If you require assistance with the installation or use of PeakVHDL, you should first visit the Accolade Design Automation Web site at URL **http://www.peakvhdl.com**. The Frequently Asked Questions section of the Product Support Page contains detailed answers to the most common questions. You will also have the opportunity to download patches and other files that may help to solve problems that you are having.

If you are unable to get the answers you need from the Web Site, you should contact your Accolade Design Automation Authorized Reseller. Your Authorized Reseller can provide the quickest response to questions related to installation and general use of the product.

If your question involves specific VHDL features, or can be best illustrated with a sample VHDL file, please send that file (or files) along with a detailed description to the following email address: **support@peakvhdl.com**. Isolating the problem to as small a VHDL source file as possible will help to speed the investigation.

In most cases, you will receive a detailed response to your emailed question within 24 hours.

Reporting Bugs

If you find what appears to be a bug in the product (such as an incompatibility with other VHDL products, or unusual product behavior), you should first check the Web Site at URL **http://www.peakvhdl.com** to see if a workaround has been described in the Frequently Asked Questions Page. Your Accolade Design Automation Authorized Reseller can also provide this information, and may be able to suggest workarounds or provide updated software.

Bug reports and feature requests are welcome via email or FAX; send your reports and requests, along with a detailed description of the problem or requested feature (please include an example, if possible) to:

Email:

support@peakvhdl.com

FAX:

(425) 739-2163

Postal Mail:

Accolade Design Automation, Inc.
550 Kirkland Way, Suite 200
Kirkland, WA 98033

Phone:

(800) 470-2686

(425) 828-2122

Appendix B: Glossary

Access Type

A data type analogous to a pointer that provides a form of data indirection. An access value is returned by an *allocator*.

Actual Parameter

An *object* or *literal* being passed as an argument to a *subprogram*, or being used as a higher-level *port* or *generic* in a *component instantiation*.

Aggregate

A form of expression that denotes the value of a *composite type* (such as an array or record). An aggregate value is specified by listing the value of each element of the aggregate, using either *positional* or *named association*.

Allocator

An operation in VHDL that creates an anonymous variable object. Allocators return *access values* that may be used to access the variable object.

Architecture

A *design unit* that describes the actual function (operation) of all or part of your design. An architecture must be associated with (bound to) an *entity*. All VHDL design descriptions must include at least one architecture.

Architecture Body

That portion of an *architecture* declaration existing between the **begin** and **end** statements of the architecture.

Array

A collection of one or more *elements* of the same type. Array data types are declared with an array *range* and an array element type, and may have more than one dimension.

Attribute

A special identifier used to return or specify information about a *named entity*. Predefined attributes are prefixed by the ‘ character. Other, non-standard attributes may be defined for specific VHDL design tools.

Base Type

The type on which a subtype is based. For example, an *array subtype* may be defined with a constrained *range*, and be based on another array type that is unconstrained.

Binding

The association of a specific *component* instance with a lower-level *entity* and *architecture*. Binding may be specified using *configuration statements* or specifications, or may be implied by the default binding.

Block

A VHDL feature allowing partitioning of the design description within an *architecture*.

Compile

The process of analyzing VHDL source file statements to create an intermediate form. In the PeakVHDL simulation environments, the compilation process results in 32-bit Windows object files that must be linked to create a *simulation executable*.

Component

An *entity* that has been declared for use in a higher-level *design entity*.

Component Declaration

A statement defining the interface (*port list* and optional *generic list*) to an entity that is to be instantiated in the current *design entity*.

Composite Type

A data type such as an *array* or *record* that includes more than one constituent *element*.

Concurrent

A characteristic of statements within the *architecture body* of a design description. Concurrent statements have no order dependency, and describe operations that are inherently parallel.

Configuration Statement

An optional design unit that specifies how a project is to be assembled prior to simulation. Configurations are somewhat akin to a parts list for your design and specify such things as the binding of *entities* to *architectures*, the mapping of *components* and *ports* to their lower-level equivalents, etc.

Constant

An declared object that has a constant value, and cannot be changed. Constants are used to give symbolic names to *literal* values, and may be declared globally (within *packages*) or locally.

Constraint

A finite range of possible values for a type or subtype.

Declared Entity

An *object*, *type*, *subprogram* or other element of the design that has been declared and given an identifying name. Declared entities have scoping, meaning that they are not visible outside the scope in which they were declared.

Declaration

A statement entered in a declarative region of the design description (such as in a *package*, or prior to the **begin** statement of an *entity*, *architecture*, *block*, *process* or *subprogram*) that creates a *declared entity*.

Delta Cycle

A simulation cycle in which all non-postponed *processes* and other *concurrent* statements are repetitively executed and *signal* assignments are schedule until no more events are pending. A complete delta cycle occurs in zero simulated time (the start and end time of the delta cycle are the same).

Descending Range

A *range* that is specified with the **downto** keyword.

Design Entity

The combination of an *entity* and its corresponding *architecture*. The minimum VHDL design description includes at least one design entity.

Design Unit

A separately compilable section of VHDL source code. The five types of design units are **entities**, *architectures*, *packages*, *package bodies* and *configurations*. Each design unit must have a unique name within the project.

Driver

The combination of a given *signal* and its current and projected future values. When two or more drivers exist for a given signal (such as when multiple values are specified for the signal at the same point in time), a *resolution function* is required.

Element

One entry in a composite type, such as an array. A one-dimensional array declared with the array bounds **0 to 7**, for example, would have eight elements.

Entity

A *design unit* that describes the interface (inputs and outputs) of all or part of your design. All VHDL design descriptions must have at least one entity declaration.

Enumeration Literal

A symbolic representation of an *enumerated type* value. Enumeration literals may take the form of either *identifiers* or characters.

Enumerated Type

A symbolic data type that is declared with an enumerated type name, and one or more enumeration values (*enumeration literals*).

Event

A change in the value of a *signal* at a given point in simulated time. Events are scheduled (rather than immediate), and always occur at the beginning of the next *delta cycle*. Events can also be delayed through the use of the **after** keyword.

Exit Condition

A expression combined an **exit** statement that specifies a condition under which a *loop* should be terminated.

Expression

A syntactically-correct sequence of *operators*, *keywords* and *literals* that defines some computed value.

Field Name

An identifier that provides access to one *element* of a *record* data type.

File Type

A data type that represents an arbitrary-length sequence of values of a given type. The most typical application of a file type is to represent a disk file, such as might be read or written during simulation.

For Loop

A *loop* construct in which the iteration scheme is a **for** statement. The **for** statement specifies a precise, finite range of the loop, and creates an index variable for the loop.

Formal Parameter

An identifier used within a *subprogram* declaration or other context in which actual *parameters*, *ports* or *generics* are to be later substituted.

Function

A *subprogram* that has a *parameter* list and returns a single value. Function parameters must be of *mode* **in**.

Generic

A *parameter* passed to an *entity*, *component* or *block* that describes additional, instance-specific information about that entity, component or block.

Global Declaration

A *declaration* that is visible to multiple *design entities*, as in the case of a declaration made within a *package*.

Hierarchy

The structure of a design description, expressed in terms of a tree of related *components*. Most simulated designs include at least two levels of hierarchy: the *test bench* and the lower-level design description.

Identifier

A sequence of characters that uniquely identify a *named entity* in a design description.

Index

A *scalar* value that specifies a precise *element*, or range of elements, within an *array*.

Infinite Loop

A *loop* that has no iteration scheme. Infinite loops should include one or more *exit* conditions to avoid endless repetitions and possibly stuck simulations.

Iteration Scheme

The starting and exit conditions for a *loop*. The iteration scheme is expressed using a **for** or **while** statement in a loop. A loop with no iteration scheme is an *infinite loop*.

Library

A storage facility allowing one or more VHDL source files (and their corresponding *design units*) to be placed in a common location. Libraries are referenced in a VHDL source file through the use of the **library** statement.

Literal

A specific value that can be applied to an *object* of a some type. Literals fall into five general catagories: bit strings, enumeration literals, numeric literals, strings, or the special literal **null**.

Loop

A sequential state providing the ability to repeat one or more statements. A loop may be finite (as in the case of a **for** loop) or infinite, depending on the nature of its *iteration scheme*.

Mode

The direction of data (either **in**, **out**, **inout** or **buffer**) of a *subprogram parameter* or *entity port*.

Named Association

A method of explicitly associating actual *parameters* and other designators with formal designators, as in a *subprogram* reference or *component instantiation*.

Named Entity

A item that has been declared and given a unique name (within the current *scope*). Examples of named entities include such things as *signals* and *variables*, *entities*, *architectures* and *blocks*, *component instances*, *processes*, *loop* labels, etc.

Object

A named entity that can be assigned (or initialized with) a value and that has a specific type. Objects include *signals*, *constants*, *variables* and *files*.

Parameter

An *object* or *literal* passed into a *subprogram* via that subprogram's parameter list.

Physical Type

A data type used to represent measurements. A physical type value is specified with an integer literal followed by a unit that has been defined for the type. One example of a standard physical type is the type **time**, which has the units **fs**, **ps**, **ns**, **us**, **ms**, **sec**, **min**, and **hr**.

Port

A interface element of an *entity* or *component*. A port must be specified with a name, data *type* and *mode*.

Positional Association

A method of specifying the mapping of *actual* and *formal parameters* (or actual and formal *ports*) by position in a list.

Procedure

A *subprogram* that has a *parameter* list and does not return a value. Procedure parameters have *modes* indicating their direction (e.g. **in**, **out**, **inout**, **buffer**).

Process

A collection of *sequential* statements that are executed whenever there is an *event* on any *signal* appearing in the process *sensitivity list* or, in the case in which there is no sensitivity list, whenever an *event* occurs (or simulation time is reached) that satifies the condition of a *wait* statement within the process. Signals assigned within a process are not updated until the current *delta cycle* is complete.

Range

A subset of the possible values of a *scaler* type. Ranges may be used, for example, to specify a *type* or *subtype* declaration, or to specify the range of a *loop*.

Record

A data type that includes more than one *element* of differing types. Record elements are identified by *field names*.

Resolution Function

A special function, associated with a *type declaration*, that describes the resulting value when two or more different values are driven onto a signal of that data type.

Scalar

A data type that has a distinct order to its values, allowing two objects or literals of that type to be compared using relational operators. Scalar types include *integers*, *enumerated types*, *floating point types*, and *physical types*.

Sequential

A characteristic of statements within *processes* and *subprograms*. Sequential statements are executed in sequence, and therefore have order dependency. Sequential statements may be used to describe sequential logic (logic that implies memory elements), or may be used to describe combinational (non-sequential) logic.

Signal

An storage object that maintains a history of *events*. Signals are created as the result of *signal declarations* or *port declarations*.

Signal Declaration

A statement that introduces (creates) a new *signal*. Signals are declared with a name and a data *type*. Signals may be declared globally (in a *package*) or locally (such as in the declarative region of an *entity*, *architecture*, or *block*). Assignments to signals are scheduled, meaning that do not have new values assigned to them until the current *delta cycle* has completed.

Slice

A one-dimensional, contiguous *array* created as a result of constraining a larger one-dimensional array.

Source File

A file containing VHDL statements describing all or part of your design.

String

A string data type is an *array* of characters. String data types may be constrained (fixed length) or unconstrained. A string *literal* is a sequence of characters enclosed by two quotation marks (“).

Subprogram

A *function* or *procedure*. Subprograms may be declared globally (in a package) or locally (such as in the declarative region of an *entity*, *architecture*, *block*, *process* or *subprogram*).

Test Bench

One or more VHDL source files describing the sequence of tests to be run on your design. Test benches are normally the top-level of a design being simulated.

Transaction

A scheduled (current or future) *event* for a given *signal*. A transaction consists of a value and a time at which that value is to be driven onto the signal.

Time Unit

A symbolic unit of time used during simulation. Time units are defined as part of the VHDL specification, and include values such as ns (nanoseconds), ms (milliseconds), etc.

Type

A declared name and a corresponding set of declared values representing the possible values of the type. Types fall into the following general catagories: *scalar types*, *composite types*, *file types*, and *access types*.

Type Conversion

An operation that results in the conversion of one data *type* to another. Type conversions may be implicit, explicit, or make use of type conversion functions.

Type Declaration

A *declaration statement* that creates a new data *type*. A type declaration must include a type name and a description of the entire set of possible values for that type.

Variable

A storage facility used in *processes* and *subprograms* to represent local data. Variables must be declared with a name and *type*. Variables are persistent with processes (meaning they retain their values in subsequent executions of the process), but are non-persistent in subprograms.

Waveform

A series of *transactions* defining the behavior of a *signal* (in terms of future *events*) over time.

Appendix C: Examples Gallery

The examples in this section are intended to help you get started with VHDL. Each example demonstrates one or more important features of the language, and demonstrates commonly used coding styles for synthesizable circuits and test benches.

These examples, and more, can be found in the \EXAMPLES subdirectory of your PeakVHDL installation. You are encouraged to copy these examples and modify them for your own use.

Using Type Conversion Functions

This example, an 8-bit counter, demonstrates one possible approach to type conversion. Type conversions are often required in VHDL due to the languages strict type checking features. In this example, a type conversion is required to convert the array data types used in the design's interface to integer data types used internally for arithmetic operations. For demonstration purposes, we are using a custom type conversion function that is defined in the design description. In most cases, you will want to use a standard type conversion function from the IEEE library, or use a type conversion function provided by your synthesis vendor.

Note: Another option when numeric values are required is to make use of the IEEE 1076.3 ***numeric_std*** *package. This package is provided in the library IEEE supplied with the PeakVHDL simulator. See example NUMERIC for more details about 1076.3 and the* ***numeric_std*** *package.*

Design Description

```
library ieee;
use ieee.std_logic_1164.all;

package conversions is
   function to_unsigned (a: std_ulogic_vector) return integer;
   function to_vector (size: integer; num: integer) return std_ulogic_vector;
end conversions;

package body conversions is
   ------------------------------------------------------
   -- Convert a std_ulogic_vector to an unsigned integer
   --
   function to_unsigned (a: std_ulogic_vector) return integer is
      alias av: std_ulogic_vector (1 to a'length) is a;
      variable ret,d: integer;
   begin
      d := 1;
      ret := 0;
      for i in a'length downto 1 loop
         if (av(i) = '1') then
```

```
            ret := ret + d;
         end if;
         d := d * 2;
      end loop;
      return ret;
   end to_unsigned;

   ------------------------------------------------------
   -- Convert an integer to a std_ulogic_vector
   --
   function to_vector (size: integer; num: integer) return std_ulogic_vector
is
      variable ret: std_ulogic_vector (1 to size);
      variable a: integer;
   begin
      a := num;
      for i in size downto 1 loop
         if ((a mod 2) = 1) then
            ret(i) := '1';
         else
            ret(i) := '0';
         end if;
         a := a / 2;
      end loop;
      return ret;
   end to_vector;

end conversions;


------------------------------------------------------
-- COUNT16: 4-bit counter.
--
Library ieee;
Use ieee.std_logic_1164.all;
use work.conversions.all;

Entity COUNT16 Is
   Port (Clk,Rst,Load: in std_ulogic;
         Data: in std_ulogic_vector(3 downto 0);
         Count: out std_ulogic_vector(3 downto 0)
   );
End COUNT16;

Architecture COUNT16_A of COUNT16 Is
Begin
```

```
    process(Rst,Clk)

    -- Note the use of a variable to localize the feedback
    -- behavior of the counter registers. This is good general
    -- design practice in VHDL, as it helps to cut down on
    -- unwanted side-effects. In this example, the use of
    -- a variable of type integer also localizes the use of
    -- a numeric data type to within the process itself. This
    -- makes it easier to modify the design as necessary when
    -- using different type conversion routines.

      variable Q: integer range 0 to 15;

    begin

      if Rst = '1' then              -- Asynchronous reset
        Q := 0;
      elsif rising_edge(Clk) then
        if Load = '1' then
          Q := to_unsigned(Data);  -- Convert vector to integer
        elsif Q = 15 then
          Q := 0;
        else
          Q := Q + 1;
        end if;
      end if;

      Count <= to_vector(4,Q);        -- Convert integer to vector
                                      -- for use outside the process.

    end process;

End COUNT16_A;
```

Test Bench

```
library ieee;
Use ieee.std_logic_1164.all;

Entity T_COUNT16 Is
End T_COUNT16;

use work.count16;
```

```
Architecture stimulus of T_COUNT16 Is
   Component COUNT16
      Port (Clk,Rst,Load: in std_ulogic;
            Data: in std_ulogic_vector(3 downto 0);
            Count: out std_ulogic_vector(3 downto 0)
      );
   End Component;
   Signal Clk,Rst,Load: std_ulogic; -- Top level signals
   Signal Data: std_ulogic_vector(3 downto 0);
   Signal Count: std_ulogic_vector(3 downto 0);
   Signal Clock_cycle: natural := 0;

Begin
   DUT: COUNT16 Port Map (Clk,Rst,Load,Data,Count);

   -- The first process sets up a 20Mhz background clock
   CLOCK: process
   begin
      Clock_cycle <= Clock_cycle + 1;
      Clk <= '1';
      wait for 25 ns;
      Clk <= '0';
      wait for 25 ns;
   end process;

   -- This process applies stimulus to reset and load the counter...
   Stimulus1: Process
   Begin
      Rst <= '1';
      wait for 40 ns;
      Rst <= '0';
      Load <= '1';
      Data <= "0100";    -- Load 0100 into the counter
      wait for 50 ns;
      Load <= '0';
      wait for 500 ns;
      Load <= '1';
      Data <= "0000";    -- Load 0000 into the counter
      wait for 50 ns;
      Load <= '0';
      wait for 11000 ns;
      wait;
   End Process;
End stimulus;
```

Using Components

This example is a structural description of a T flip-flop counter. The example demonstrates the use of component port maps and default interfaces.

Design Description

```
------------------------------------------------------------
-- TCOUNT.VHD
--
library ieee;
use ieee.std_logic_1164.all;

entity andgate is
   port(A,B,C,D: in std_ulogic := '1';
        Y: out std_ulogic);
end andgate;

architecture gate of andgate is
begin
   Y <= A and B and C and D;
end gate;

use ieee.std_logic_1164.all;
entity tff is
   port(Rst,Clk,T: in std_ulogic;
        Q: out std_ulogic);
end tff;

architecture behavior of tff is
begin
   process(Rst,Clk)
      variable Qtmp: std_ulogic;
   begin
      if (Rst = '1') then
         Qtmp := '0';
      elsif rising_edge(Clk) then
         if T = '1' then
            Qtmp := not Qtmp;
         end if;
      end if;
      Q <= Qtmp;
```

```
   end process;
end behavior;

use ieee.std_logic_1164.all;
entity TCOUNT is
   port (Rst: in std_ulogic;
         Clk: in std_ulogic;
         Count: out std_ulogic_vector(4 downto 0)
   );
end TCOUNT;

architecture STRUCTURE of TCOUNT is
   component tff
      port(Rst,Clk,T: in std_ulogic;
           Q: out std_ulogic);
   end component;
   component andgate
      port(A,B,C,D: in std_ulogic := '1';
           Y: out std_ulogic);
   end component;
   constant VCC: std_ulogic := '1';
   signal T,Q: std_ulogic_vector(4 downto 0);
begin

   T(0) <= VCC;
   T0: tff port map (Rst=>Rst, Clk=>Clk, T=>T(0), Q=>Q(0));
   T(1) <= Q(0);
   T1: tff port map (Rst=>Rst, Clk=>Clk, T=>T(1), Q=>Q(1));
   A1: andgate port map(A=>Q(0), B=>Q(1), Y=>T(2));
   T2: tff port map (Rst=>Rst, Clk=>Clk, T=>T(2), Q=>Q(2));
   A2: andgate port map(A=>Q(0), B=>Q(1), C=>Q(2), Y=>T(3));
   T3: tff port map (Rst=>Rst, Clk=>Clk, T=>T(3), Q=>Q(3));
   A3: andgate port map(A=>Q(0), B=>Q(1), C=>Q(2), D=>Q(3), Y=>T(4));
   T4: tff port map (Rst=>Rst, Clk=>Clk, T=>T(4), Q=>Q(4));

   Count <= Q;

end STRUCTURE;
```

Test Bench

```
------------------------------------------------------------
-- Auto-generated test bench for TCOUNT
--
library ieee;
```

```
use ieee.std_logic_1164.all;
use std.textio.all;
use work.TCOUNT;

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is
component TCOUNT
   port (
      Rst: in std_ulogic;
      Clk: in std_ulogic;
      Count: out std_ulogic_vector(4 downto 0)
   );
end component;
constant PERIOD: time := 100 ns;
-- Top level signals go here...
signal Rst: std_ulogic;
signal Clk: std_ulogic;
signal Count: std_ulogic_vector(4 downto 0);
signal done: boolean := false;

for DUT: TCOUNT use entity work.TCOUNT(STRUCTURE);

begin
   DUT: TCOUNT port map (
         Rst => Rst,
         Clk => Clk,
         Count => Count
       );

   CLOCK1: process
      variable clktmp: std_ulogic := '0';
   begin
      wait for PERIOD/2;
      clktmp := not clktmp;
      Clk <= clktmp;  -- Attach your clock here
      if done = true then
         wait;
      end if;
   end process;

   STIMULUS1: process
   begin

      Rst <= '1';
```

```
        wait for PERIOD;

        Rst <= '0';

        wait for PERIOD * 36;

        done <= true;
        wait;
    end process;

end stimulus;
```

Using Generate Statements

This example demonstrates the use of **generate** statements. The parity generation circuit is built from a chain of exclusive-OR gates, which have been defined separately.

Design Description

```
-------------------------------------------------
-- Parity generator.
--
library ieee;
use ieee.std_logic_1164.all;

entity parity10 is
   port(D: in std_ulogic_vector(0 to 9);
        ODD: out std_ulogic);
   constant WIDTH: integer := 10;
end parity10;

library gates;
use gates.xor2;

architecture structure of parity10 is
   component xor2
      port(A,B: in std_ulogic;
           Y: out std_ulogic);
   end component;

   signal p: std_ulogic_vector(0 to WIDTH - 2);

   for G: xor2 use entity gates.xor2(xor_a);

begin
   -- The outermost generate loop is a for-generate
   -- that repeats once for each of the XOR gates required
   -- for the circuit...
   G: for I in 0 to (WIDTH - 2) generate

      -- This generate statement creates the first XOR gate
      -- in the series...

      G0: if I = 0 generate
```

```
            X0: xor2 port map(A => D(0), B => D(1), Y => p(0));
        end generate G0;

        -- This generate statement creates the middle XOR gates
        -- in the series...

        G1: if I > 0 and I < (WIDTH - 2) generate
            X0: xor2 port map(A => p(i-1), B => D(i+1), Y => p(i));
        end generate G1;

        -- This generate statement creates the last XOR gate
        -- in the series...

        G2: if I = (WIDTH - 2) generate
            X0: xor2 port map(A => p(i-1), B => D(i+1), Y => ODD);
        end generate G2;

    end generate G;

end structure;
```

Test Bench

```
library ieee;
use ieee.std_logic_1164.all;

entity testbnch is
end testbnch;

use work.parity10;

architecture behavior of testbnch is
    component parity10
        port(D: in std_ulogic_vector(0 to 9);
            ODD: out std_ulogic);
    end component;
    signal D: std_ulogic_vector(0 to 9);
    signal ODD: std_ulogic;

begin

    DUT: parity10 port map (D,ODD);
```

```
    process
    begin
        D <= "0000000001";
        wait for 50 ns;
        D <= "1000000001";
        wait for 50 ns;
        D <= "0100100001";
        wait for 50 ns;
        D <= "0000000011";
        wait for 50 ns;
        D <= "0100000000";
        wait for 50 ns;
        D <= "1010100010";
        wait for 50 ns;
        D <= "1111111101";
        wait for 50 ns;
        D <= "0111000001";
        wait for 50 ns;
        D <= "1000000000";
        wait for 50 ns;
    end process;
end behavior;
```

Understanding Sequential Signal Assignments

This CRC generator implements the CRC-CCITT standard for serial data transmission. The VHDL description for the CRC generator is based in an ABEL design appearing in "Digital Design Using ABEL" by David Pellerin and Michael Holley (Prentice Hall, 1994).

This design description demonstrates the important distinction between signals and variables in VHDL. Note that the chain of registers and XOR operations has been written using signals within a process. It would be possible to describe the same circuit using variables, but great care would have to be taken to ensure that the desired circuit is produced. This is because variable assignments are immediate: writing variable assignments such as

```
--          X(0)  := Din xor X(15);
--          X(1)  := X(0);
--          X(2)  := X(1);
--          X(3)  := X(2);
--          X(4)  := X(3);
```

would *not* result in a single XOR function and a simple chain of registers. Instead, the expression 'Din xor X(15)' would be assumed as the input to all registers, rather than just X(0).

Design Description

```
-------------------------------------------
-- 8-bit Serial CRC Generator.
--
library ieee;
use ieee.std_logic_1164.all;

entity crc8s is
  port (Clk,Set, Din: in std_ulogic;
        CRC_Sum: out std_ulogic_vector(15 downto 0));
end crc8s;
```

```
architecture behavior of crc8s is
   signal X: std_ulogic_vector(15 downto 0);
begin
   process(Clk,Set)
   begin
      if Set = '1' then
         X <= (others=> '1');
      elsif rising_edge(Clk) then
         X(0)  <= Din xor X(15);
         X(1)  <= X(0);
         X(2)  <= X(1);
         X(3)  <= X(2);
         X(4)  <= X(3);
         X(5)  <= X(4) xor Din xor X(15);
         X(6)  <= X(5);
         X(7)  <= X(6);
         X(8)  <= X(7);
         X(9)  <= X(8);
         X(10) <= X(9);
         X(11) <= X(10);
         X(12) <= X(11) xor Din xor X(15);
         X(13) <= X(12);
         X(14) <= X(13);
         X(15) <= X(14);
      end if;

   end process;

   CRC_Sum <= X;

end behavior;
```

Test Bench

This test bench applies a sequence of values to the CRC generator, and demonstrates the use of record data types.

```
-------------------------------------------------------
-- Test bench for CRC generator
--

library ieee;
use ieee.std_logic_1164.all;

use work.crc8s;    -- Get the design out of library 'work'
```

```
entity testcrc is
end testcrc;

architecture stimulus of testcrc is
   component crc8s
      port (Clk,Set,Din: in std_ulogic;
            CRC_Sum: out std_ulogic_vector(15 downto 0));
   end component;

   signal CE: std_ulogic;
   signal Clk,Set: std_ulogic;
   signal Din: std_ulogic;
   signal CRC_Sum: std_ulogic_vector(15 downto 0);
   signal vector_cnt: integer := 1;
   signal error_flag: std_ulogic := '0';

   type test_record is record
      CE      : std_ulogic;  -- Clock enable
      Set     : std_ulogic;
      Din     : std_ulogic;
      CRC_Sum : std_ulogic_vector (15 downto 0);
   end record;

   type test_array is array(positive range <>) of test_record;

   signal svector: test_record;

   constant test_vectors : test_array := (
 --  CE, Set, Din,      CRC_Sum
            ('0', '1', '0', "----------------"),  -- Reset

            ('1', '0', '0', "----------------"), -- 'H'
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '0', "0010100000111100"), -- 0x283C

            ('1', '0', '0', "----------------"), -- 'e'
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
```

```
            ('1', '0', '1', "1010010101101001"), -- 0xA569

            ('1', '0', '0', "----------------"), -- 'l'
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '0', "0010000101100101"), -- 0x2165

            ('1', '0', '0', "----------------"), -- 'l'
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '0', "1111110001101001"), -- 0xFC69

            ('1', '0', '0', "----------------"), -- 'o'
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '0', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "----------------"),
            ('1', '0', '1', "1101101011011010")  -- 0xDADA
    );

begin
    -- instantiate the component
    DUT: crc8s port map(Clk,Set,Din,CRC_Sum);

    -- provide Stimulus and check the result

    testrun: process
        variable vector : test_record;
    begin
        for index in test_vectors'range loop
            vector_cnt <= index;
            vector := test_vectors(index);
            svector <= vector;  -- so we can see it in simulation

            -- Apply the input stimulus...
            CE <= vector.CE;
```

```
            Set <= vector.Set;
            Din <= vector.Din;

             -- Clock (low-high-low) with a 100 ns cycle...
            Clk <= '0';
            wait for 25 ns;
            if CE = '1' then
               Clk <= '1';
            end if;
            wait for 50 ns;
            Clk <= '0';
            wait for 25 ns;

            -- Check the results...
            if (vector.CRC_Sum /= "----------------"
                  and CRC_Sum  /= vector.CRC_Sum) then
               error_flag <= '1';
               assert false
                  report "Output did not match!"
                  severity WARNING;
            else
               error_flag <= '0';
            end if;
         end loop;
         wait;
      end process;

end stimulus;
```

Describing A State Machine

This example demonstrates how to write a synthesizable state machine description using processes and enumerated types.

The circuit, a video frame grabber controller, was first described in *Practical Design Using Programmable Logic* by David Pellerin and Michael Holley (Prentice Hall, 1990). A slightly modified form of the circuit also appears in the *ATMEL Configurable Logic Design and Application Book*, 1993-1994 edition.

The circuit described is a simple freeze-frame unit that 'grabs' and holds a single frame of NTSC color video image. This design description includes the frame detection and capture logic. The complete circuit requires an 8-bit D-A / A-D converter and a 256K X 8 static RAM.

Design Description

```
------------------------------------------------------------
-- A Video Frame Grabber.
--
Library ieee;
Use ieee.std_logic_1164.all;

Entity CONTROL Is
   Port (Reset: in std_ulogic;
      Clk: in std_ulogic;
      Mode: in std_ulogic;
      Data: in std_ulogic_vector(7 downto 0);
      TestLoad: in std_ulogic;
      Addr: out integer range 0 to 253243;
      RAMWE: out std_ulogic;
      RAMOE: out std_ulogic;
      ADOE: out std_ulogic );
End CONTROL;

Architecture CONTROL_A of CONTROL Is
   constant FRAMESIZE: integer := 253243;
```

```
   constant TESTADDR: integer := 253000;

   signal ENDFR: std_ulogic;
   signal INCAD: std_ulogic;
   signal VS: std_ulogic;
   signal Sync: integer range 0 to 127;
   type states is (StateLive,StateWait,StateSample,StateDisplay);
   signal current_state, next_state: states;
Begin

   -- Address counter. This counter increments until we reach the end of
   -- the frame (address 253243), or until the input INCAD goes low.

   ADDRCTR: process(Clk)
      variable cnt: integer range 0 to FRAMESIZE;
   begin
      if rising_edge(Clk) then
         if TestLoad = '1' then
            cnt := TESTADDR;
            ENDFR <= '0';
         else
            if INCAD = '0' or cnt = FRAMESIZE then
               cnt := 0;
            else
               cnt := cnt + 1;
            end if;
            if cnt = FRAMESIZE then
               ENDFR <= '1';
            else
               ENDFR <= '0';
            end if;
         end if;
      end if;
      Addr <= cnt;
   end process;

   -- Vertical sync detector. Here we look for 128 bits of zero, which
   -- indicates the vertical sync blanking interval.

   SYNCCTR: process(Reset,Clk)
   begin
      if Reset = '1' then
         Sync <= 0;
      elsif rising_edge(Clk) then
         if Data /= "00000000" or Sync = 127 then
            Sync <= 0;
```

```
        else
            Sync <= Sync + 1;
        end if;
    end if;
end process;

VS <= '1' when Sync = 127 else '0';

-- State register process:

STREG: process(Reset,Clk)
begin
    if Reset = '1' then
        current_state <= StateLive;
    elsif rising_edge(Clk) then
        current_state <= next_state;
    end if;
end process;

-- State transitions:

STTRANS: process(current_state,Mode,VS,ENDFR)
begin
    case current_state is
        when StateLive =>    -- Display live video on the output
            RAMWE <= '1';
            RAMOE <= '1';
            ADOE <= '0';
            INCAD <= '0';
            if Mode = '1' then
                next_state <= StateWait;
            end if;
        when StateWait =>    -- Wait for vertical sync
            RAMWE <= '1';
            RAMOE <= '1';
            ADOE <= '0';
            INCAD <= '0';
            if VS = '1' then
                next_state <= StateSample;
            end if;
        when StateSample =>  -- Sample one frame of video
            RAMWE <= '0';
            RAMOE <= '1';
            ADOE <= '0';
            INCAD <= '1';
            if ENDFR = '1' then
```

```
            next_state <= StateDisplay;
          end if;
        when StateDisplay => -- Display the stored frame
          RAMWE <= '1';
          RAMOE <= '0';
          ADOE <= '1';
          INCAD <= '1';
          if Mode = '1' then
            next_state <= StateLive;
          end if;
      end case;
    end process;

End CONTROL_A;
```

Test Bench

The following test bench uses loops to simplify the description of a long test sequence.

```
library ieee;
Use ieee.std_logic_1164.all;
Use std.textio.all;

library work;
use work.control;

Entity T_CONTROL Is
End T_CONTROL;

Architecture stimulus of T_CONTROL Is
Component CONTROL
   Port (Reset: in std_ulogic;
      Clk: in std_ulogic;
      Mode: in std_ulogic;
      Data: in std_ulogic_vector(7 downto 0);
      TestLoad: in std_ulogic;
      Addr: out integer range 0 to 253243;
      RAMWE: out std_ulogic;
      RAMOE: out std_ulogic;
      ADOE: out std_ulogic);
End Component;
Constant PERIOD: time := 100 ns;
```

```
-- Top level signals go here...
Signal Reset: std_ulogic;
Signal Clk: std_ulogic;
Signal Mode: std_ulogic;
Signal Data: std_ulogic_vector(7 downto 0);
Signal TestLoad: std_ulogic;
Signal Addr: integer range 0 to 253243;
Signal RAMWE: std_ulogic;
Signal RAMOE: std_ulogic;
Signal ADOE:  std_ulogic;
Signal done: boolean := false;

Begin
   DUT: CONTROL Port Map (
         Reset => Reset,
         Clk => Clk,
         Mode => Mode,
         Data => Data,
         TestLoad => TestLoad,
         Addr => Addr,
         RAMWE => RAMWE,
         RAMOE => RAMOE,
         ADOE => ADOE
       );

   Clock1: process
      variable clktmp: std_ulogic := '0';
   begin
      wait for PERIOD/2;
      clktmp := not clktmp;
      Clk <= clktmp;  -- Attach your clock here
      if done = true then
         wait;
      end if;
   end process;

   Stimulus1: Process
   Begin
      -- Sequential stimulus goes here...
      Reset <= '1';
      Mode <= '0';
      Data <= "00000000";
      TestLoad <= '0';
      wait for PERIOD;
      Reset <= '0';
      wait for PERIOD;
```

```
        Data <= "00000001";
        wait for PERIOD;
        Mode <= '1';

        -- Check to make sure we detect the vertical sync...
        Data <= "00000000";
        for i in 0 to 127 loop
           wait for PERIOD;
        end loop;

        -- Now sample data to make sure the frame counter works...
        Data <= "01010101";
        for i in 0 to 100000 loop
           wait for PERIOD;
        end loop;

        -- Load in the test value to check the end of frame detection...
        TestLoad <= '1';
        wait for PERIOD;
        TestLoad <= '0';
        for i in 0 to 300 loop
           wait for PERIOD;
        end loop;
        done <= true;

     End Process;

End stimulus;
```

Reading And Writing From Files

More complex test benches often make use of VHDL's file read and write capabilities. These features make it easy to create test benches that operate on data stored in files, such as test vectors. The following example demonstrates how you can use the text I/O features of VHDL to read test data from an ASCII file.

Consider a Fibonacci sequence generator. A Fibonnaci sequence is a series of numbers, beginning with 1, 1, 2, 3, 5..., in which every number in the sequence is the sum of the previous two numbers. To construct a circuit that generates an *n*-bit Fibonacci sequence, two *n*-bit registers — **A** and **B** — are required to store the last two values of the sequence and add them to produce the next value.

To initialize the circuit, the **A** and **B** registers must be loaded with values of 0 and 1 respectively. Subsequent cycles of the circuit must move the calculated next value into the **B** register while moving the value stored in the **B** register to the **A** register. In this implementation, the **A** and **B** registers form a 2-deep first-in first-out (FIFO) stack.

The VHDL source file shown below describes this Fibonnaci sequence generator.

Design Description

```
--------------------------------------------------------------------------------
-- Fibonacci sequence generator.
--
-- Copyright 1996, Accolade Design Automation, Inc.
--
library ieee;
use ieee.std_logic_1164.all;

entity fib is
  port (Clk,Clr: in std_ulogic;
        Load: in std_ulogic;
        Data_in: in std_ulogic_vector(15 downto 0);
```

```
        S: out std_ulogic_vector(15 downto 0));
end fib;

architecture behavior of fib is
   signal Restart,Cout: std_ulogic;
   signal Stmp: std_ulogic_vector(15 downto 0);
   signal A, B, C: std_ulogic_vector(15 downto 0);
   signal Zero: std_ulogic;
   signal CarryIn, CarryOut: std_ulogic_vector(15 downto 0);

begin
   P1: process(Clk)
   begin
      if rising_edge(Clk) then
          Restart <= Cout;
      end if;
   end process;

   Stmp <= A xor B xor CarryIn;
   Zero <= '1' when Stmp = "0000000000000000" else '0';

   CarryIn <= C(15 downto 1) & '0';
   CarryOut <= (B and A) or ((B or A) and CarryIn);
   C(15 downto 1) <= CarryOut(14 downto 0);
   Cout <= CarryOut(15);

   P2: process(Clk,Clr,Restart)
   begin
      if Clr = '1' or Restart = '1' then
         A <= "0000000000000000";
         B <= "0000000000000000";
      elsif rising_edge(Clk) then
         if Load = '1' then
            A <= Data_in;
         elsif Zero = '1' then
            A <= "0000000000000001";
         else
            A <= B;
         end if;
         B <= Stmp;
      end if;
   end process;

   S <= Stmp;

end behavior;
```

Test Bench

The following test bench reads lines from an ASCII file and applies the data contained in each line as a test vector to stimulate and test the Fibonacci circuit.

```
-- Test bench for Fibonacci sequence generator.

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;              -- Use the text I/O features of the standard library
use work.fib;                    -- Get the design out of library 'work'

entity testfib is                -- Entity; once again we have no ports
end testfib;

architecture stimulus of testfib is
   component fib                 -- Create one instance of the fib design unit
     port (Clk,Clr: in std_ulogic;
           Load: in std_ulogic;
           Data_in: in std_ulogic_vector(15 downto 0);
           S: out std_ulogic_vector(15 downto 0));
   end component;

   -- The following conversion functions are used to process the test data
   -- and convert from string data to array data...
   function str2vec(str: string) return std_ulogic_vector is
   variable vtmp: std_ulogic_vector(str'range);
   begin
      for i in str'range loop
         if (str(i) = '1') then
            vtmp(i) := '1';
         elsif (str(i) = '0') then
            vtmp(i) := '0';
         else
            vtmp(i) := 'X';
         end if;
      end loop;
      return vtmp;
   end;
```

```
   function vec2str(vec: std_ulogic_vector) return string is
   variable stmp: string(vec'left+1 downto 1);
   begin
      for i in vec'reverse_range loop
         if (vec(i) = '1') then
            stmp(i+1) := '1';
         elsif (vec(i) = '0') then
            stmp(i+1) := '0';
         else
            stmp(i+1) := 'X';
         end if;
      end loop;
   return stmp;
   end;


   signal Clk,Clr: std_ulogic;                    -- Declare local signals
   signal Load: std_ulogic;
   signal Data_in: std_ulogic_vector(15 downto 0);
   signal S: std_ulogic_vector(15 downto 0);
   signal done: std_ulogic := '0';


   constant PERIOD: time := 50 ns;

   for DUT: fib use entity work.fib(behavior);          -- Configuration
                                                        -- specification
begin
   DUT: fib port map(Clk=>Clk,Clr=>Clr,Load=>Load,      -- Creates one
            Data_in=>Data_in,S=>S);                     -- instance

   Clock: process
      variable c: std_ulogic := '0';     -- Background clock process
   begin
      while (done = '0') loop            -- The done flag indicates that we
         wait for PERIOD/2;              -- are finished and can stop the clock.
         c := not c;
         Clk <= c;
      end loop;
   end process;


   read_input: process
      file vector_file: text is in "testfib.vec";      -- File declaration

      variable stimulus_in: std_ulogic_vector(33 downto 0);      -- Temporary
```

```
                                                    -- storage
                                                    -- for inputs

    variable S_expected: std_ulogic_vector(15 downto 0);    -- Temporary
                                                    -- storage
                                                    -- for
                                                    -- outputs

    variable str_stimulus_in: string(34 downto 1);    -- Temporary storage
                                                      -- for big string

    variable err_cnt: integer := 0;          -- Keeps track of how many errors
    variable file_line: line;                -- Text line buffer; 'line' is a
                                             -- standard type (textio library).

begin
    wait until rising_edge(Clk);             -- Synchronizes with first clock

    while not endfile(vector_file) loop      -- Loops through the lines in
                                             -- the file

      readline (vector_file,file_line);      -- Reads one complete line
                                             -- into file_line

      read (file_line,str_stimulus_in) ;     -- Extracts the first field from
                                             -- file_line

      stimulus_in := str2vec(str_stimulus_in);    -- Converts the input
                                                  -- string to a vector

      wait for 1 ns;                         -- Delays for a nanosecond

      Clr <= stimulus_in(33);                     -- Gets each input's
      Load <= stimulus_in(32);                    -- value from the test
      Data_in <= stimulus_in(31 downto 16);       -- vector array and
                                                  -- assigns the values


      S_expected := stimulus_in(15 downto 0);

      wait until falling_edge(Clk);          -- Waits until the clock goes
                                             -- back to '0' (midway through
                                             -- the clock cycle)


      if (S /= S_expected) then
```

```
            err_cnt := err_cnt + 1;        -- Increments the error counter and
            assert false                   -- reports an error if different
              report "Vector failure!" & lf &
              "Expected S to be  " & vec2str(S_expected) & lf &
              "but its value was " & vec2str(S) & lf
              severity note;
          end if;
        end loop;                          -- Continues looping through the file

        done <= '1';                       -- Sets a flag when we are finished; this
                                           -- will stop the clock.

        wait;                              -- Suspends the simulation

    end process;

end stimulus;
```

Appendix D: SV/OLE Reference

This chapter describes the programming interface to the SV/OLE server. The purpose of this chapter is to allow sophisticated users direct access to, and control over, all aspects of the SV/OLE simulation server.

This chapter assumes that you have a basic understanding of OLE Automation in the Windows environment, and assumes that you are familiar with Microsoft's Visual Basic or Visual C++ development systems. Before using this chapter, you should study the sample Visual Basic application presented in the previous chapter.

The example code in this chapter is written in Visual Basic (version 4), and does not necessarily reflect the actual code you will write when interfacing to SV/OLE. Specifically, the examples shown include minimal (if any) error checking to ensure that the OLE Automation calls have properly executed. Errors can occur during these calls for various reasons, such as a lack of memory or resources, or incorrect argument data. Errors in the SVOLE server itself can also result in OLE Automation errors. Refer to your Visual Basic or Visual C++ documentation and the examples provided in the previous chapter (and in the **examples\svole** directory), for information about detecting and recovering from such errors.

SV/OLE Operation Overview

The SV / OLE server is an OLE-enabled Windows application that is used to load and execute previously compiled and linked VHDL design descriptions, called *simulation executables*.

When you install PeakVHDL, the SV / OLE server is registered with Windows under the name **Accolade.Sim.4**.

Note: The version number appended to theend of the server name is updated on a regular basis. To determine the most current server name, examine the Windows Registry (using REGEDIT) and search for the ***Accolade.Sim*** *entry in the* ***Classes*** *section of the* ***HKEY_LOCAL_MACHINE/Software*** *registry section .*

In the following sections you will find detailed descriptions of all of the methods and properties available from the *default dispatch* of the SV / OLE server. Before using these methods and properties, you must first create an *instance* of the SV / OLE server.

To create an instance of the SV / OLE server, you will issue a call to the OLE system to create the SV / OLE server object. In Visual Basic, you would use the **CreateObject()** function as follows:

```
dim SVOLE as object

Set  SVOLE = CreateObject("Accolade.Sim.4")
```

Note: It is important to check the status ***SVOLE*** *at this point to ensure that the OLE server object has been created. In Visual Basic, you must check the* ***Err*** *flag to determine if the call to* ***CreateObject()*** *was successful.*

The general sequence of operation for the **Accolade.Sim.4** SV / OLE server is:

1. Create the OLE server object using **CreateObject** (or other documented method, depending on your software development environment).

2. Set the arguments of SV/OLE with the **Args()** method. The **Args()** method accepts a simulation executable file name as its argument and causes that simulation executable to be loaded.
3. Set up a timer or other polling system to periodically check the status of the running SV/OLE process using the **QueryStatus()** and **QueryPercentDone()** methods.
4. Using that same timer, periodically collect transcript messages and display them using **GetTranscriptText()**.
5. Set up the simulation (possibly based on user-supplied inputs) using **GetVariables()**, **GetUserTypes()**, **AddWatch()**, **DeleteWatch()**, **TimeStep()**, etc.
6. Start the simulation running with **Start()**.
7. While the simulation runs, periodically check the status of SV/OLE with **QueryStatus()** and handle assertion and Text I/O conditions as needed.
8. If necessary, halt a running simulation (typically when requested by the user) using the **Stop()** method.
9. Check for simulation completion with the **QueryStatus()** method.
10. Query the event database using **GetEvents()** to display or process the simulation event data.
11. Remove the OLE server object from memory using the **Exit()** method.

For specific examples of how to use these methods and properties, refer to the tutorial presented in the preceding chapter.

FUNCTION SVOLE.Args()

INTERFACE

FUNCTION Args(String) return Boolean

DESCRIPTION

Sets the arguments for SV/OLE. If the arguments are valid, the simulation executable will be loaded and initialization will be started before the **Args()** function returns. The SV/OLE server will be in the **RUNNING** state (see **QueryStatus**) upon return, indicating that the initialization phase has been started. During initialization the initial value of every signal driver is computed and each implicit and user defined process is executed once, as per the standard. During this initialization procedure, any activity that might occur during the simulation of the design could potentially occur, and your interface must be prepared to handle this. These activities include things like interactive input/output with the text I/O predefined files **INPUT** and **OUTPUT**, transcript messages, etc. The end of the initialization phase is complete when SVOLE enters the **READY** state (see **QueryStatus**).

INPUTS

The argument list. The argument list is a string in the following format:

simfile [-g]

Valid arguments are:

simfile	Specifies the name of the compiled and linked simulation executable.
-g	Insert source-level debugging information

OUTPUTS

True, if successful.

CONDITIONS

This method may only be called once, and must be the first method called.

EXAMPLE

```
Set SVOLE = CreateObject("Accolade.Sim.1")
argument$ = "myfile -g"
OLEStatus = SVOLE.Args(argument$)
OLEStatus = SVOLE.Start
```

FUNCTION SVOLE.AtomToName()

INTERFACE

```
FUNCTION AtomToName(Long) return String
```

DESCRIPTION

Variable and signal names are each given a unique integer to represent them for more efficient internal communication and in some of the more frequently used OLE methods. This method converts a integer back into the signal / variable name that it has been assigned to.

INPUTS

An integer that has been assigned to a signal or variable name.

OUTPUTS

The name which the given integer is assigned to.

CONDITIONS

None

EXAMPLE

```
Dim SigName as string
SigName$ = SVOLE.AtomToName(Atom)
```

FUNCTION SVOLE.CurrentBP()

INTERFACE

```
FUNCTION CurrentBP() return String
```

DESCRIPTION

Reports the VHDL source location of the next statement of VHDL code that will be executed. If the simulation has stopped at a user break point, then this is the same as the location of the actual break point that was set. It is possible, however, for the simulation to stop at a point other than the specified break point. This is because additional executable code is created by the PeakVHDL compiler to support SV/OLE, ane this code does not correspond to any actual VHDL source code. In this case a negative value is returned for the source line number. A negative value indicates that the simulation is not stopped on a specific (viewable) source file line. The client application may at this point choose to repeatedly call the **SingleStep()** method until it reaches code that corresponds to user written (viewable) VHDL code.

INPUTS

None

OUTPUTS

Returns the VHDL source location in the form “module-name:line-number” as in “adder:32”, unless the next line of code does not correspond to a user-written code. In the latter case a negative value is returned.

CONDITIONS

A design must have been previously loaded with the **Args()** method and the kernel must be in the **READY** state (see **QueryStatus**).

EXAMPLE

```
Dim colon as integer
Dim ModuleName as string
Dim LineNumber as integer
BPString$ = SVOLE.CurrentBP
colon% = Instr(BPString$,":")
ModuleName$ = Left$(BPString$, colon% - 1)
LineNumber% = Val(Mid$(BPString$, colon% + 1)
```

FUNCTION SVOLE.DeleteBP()

INTERFACE

```
FUNCTION DeleteBP(String) return Boolean
```

DESCRIPTION

Deletes a previously set break point.

INPUTS

A string specifying the location of the ACTUAL break point. This should be the string returned by the **SetBP()** function, when the break point was set.

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method and the kernel must be in the **READY** state (see **QueryStatus**).

EXAMPLE

```
OLEStatus = SVOLE.DeleteBP(BPString$)
```

FUNCTION SVOLE.DeltaStep()

INTERFACE

```
FUNCTION DeltaStep() return Boolean
```

DESCRIPTION

Starts the simulation for one delta of simulation. This corresponds to one simulation step, as defined by the VHDL standard.

INPUTS

None

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method and SV/OLE must be in the **READY** state (see **QueryStatus**).

EXAMPLE

```
OLEStatus = SVOLE.DeltaStep
```

FUNCTION SVOLE.Exit()

INTERFACE

```
FUNCTION Exit() return Boolean
```

DESCRIPTION

Terminates the OLE server.

INPUTS

None

OUTPUTS

True, if successful.

CONDITIONS

May not operate correctly if SV/OLE is in the **RUNNING** state. Use the **Stop()** method before invoking this method when in the **RUNNING** state.

EXAMPLE

```
Case “Exit”
   OLEStatus = SVOLE.Exit
```

```
timerStrobe.Enabled = False
```

FUNCTION SVOLE.GetAssertion()

INTERFACE

```
FUNCTION GetAssertion() return String
```

DESCRIPTION

Returns the text of an assertion message. This method must be called by the interface whenever SV/OLE enters the **ASSERTION_PENDING** state. See also **GetMessage**.

INPUTS

None

OUTPUTS

The text of the assertion message, if successful.

CONDITIONS

SV/OLE must be in the **ASSERTION_PENDING** state.

EXAMPLE

```
Case PROCESS_ASSERTION_PENDING
   Do While ProcessStatus = PROCESS_ASSERTION_PENDING
      sText$ = SVOLE.GetAssertion
      Call UpdateTranscript("Assertion message: " & sText$)
      DoEvents
      ProcessStatus = SVOLE.QueryStatus()
   Loop
```

FUNCTION SVOLE.GetLastError()

INTERFACE

```
FUNCTION GetLastError() return Long
```

DESCRIPTION

Returns the error code of the unsuccessful method call, or the **OK** error code, if no errors have occurred.

INPUTS

None

OUTPUTS

The error code of the unsuccessful method call, or the **OK** error code, if no errors have occurred.

CONDITIONS

None

EXAMPLE

```
if (OLEStatus = False) then
   ErrorCode = SVOLE.GetLastError     'Look for an error code
```

FUNCTION SVOLE.GetMessage()

INTERFACE

```
FUNCTION GetMessage() return String
```

DESCRIPTION

Returns the text of an assertion, user output, or VHDL run-time error message respectively when SV/OLE is in the state **ASSERTION_PENDING**, **OUTPUT_PENDING**, or **HALTED**. Can be used in place of **GetAssertion** and **GetOutput**.

INPUTS

None

OUTPUTS

String value representing the output or assertion message.

CONDITIONS

SV/OLE must be in one of the states **ASSERTION_PENDING**, **OUTPUT_PENDING**, or **HALTED.**

EXAMPLE

```
ProcessStatus = SVOLE.QueryStatus()
Select Case ProcessStatus
   Case 3:      'ASSERTION_PENDING
          . . .
      TranMsg$ = SVOLE.GetMessage
```

FUNCTION SVOLE.GetOutput()

INTERFACE

```
FUNCTION GetOutput() return String
```

DESCRIPTION

Returns the text of an user output message. Either this method or the **GetMessage** method must be called by the interface whenever SV/OLE enters the **OUTPUT_PENDING** state. This state results from a call to the IEEE standard **writeline()** procedure with a file pointer of **OUTPUT**. See also **GetMessage**.

INPUTS

None

OUTPUTS

The text of the output, if successful.

CONDITIONS

SV/OLE must be in the **OUTPUT_PENDING** state.

EXAMPLE

```
ProcessStatus = SVOLE.QueryStatus()
Select Case ProcessStatus
   Case 5:      'OUTPUT_PENDING
          . . .
      TranMsg$ = SVOLE.GetOutput
```

FUNCTION SVOLE.GetSimHist()

INTERFACE

FUNCTION GetSimHist() return SimHist Object

DESCRIPTION

This method returns an SV/OLE object (dispatch) that provides the ability to record events on selected signals/variables during the simulation. The interface to the **SimHist** server is described later in this chapter.

INPUTS

None

OUTPUTS

The OLE object.

CONDITIONS

None

EXAMPLE

```
Dim SimHist as object
Set SimHist = SVOLE.GetSimHist
```

FUNCTION SVOLE.GetTranscriptText()

INTERFACE

```
FUNCTION GetTranscriptText() return String
```

DESCRIPTION

The *transcript* contains buffered messages produced by SV/OLE. The **GetTranscript** method is called repeatedly to retrieve the buffered transcript messages. Each successive call returns the next line of the transcript. If no more transcript is currently available, an empty string is returned. The method

can be called while SV/OLE is in state **RUNNING**, and will return the transcript messages as they are produced. This asynchronous polling system implies that the **GetTranscript** call may return an empty string one time and some text at a later time, if SV/OLE has produced any transcript messages since the previous call. Also, note that transcript messages are produced during initialization of a design and upon fatal errors.

INPUTS

None

OUTPUTS

The next line of the transcript, or the empty string if no more transcript messages are available.

CONDITIONS

None

EXAMPLE

```
‘Empty the transcript buffer...
TranMsg$ = SVOLE.GetTranscriptText
Do While Len(TranMsg$)
   DoEvent
   Call UpdateTranscript(TranMsg$)
   TranMsg$ = SVOLE.GetTranscriptText
Loop
```

FUNCTION SVOLE.GetUserTypes()

INTERFACE

FUNCTION GetUserTypes() return String

DESCRIPTION

Returns the definitions of all user-defined types declared in the loaded design.

INPUTS

None

OUTPUTS

A string containing the list of definitions as defined by the following grammar:

```
definition-list: {definition "\n"}
definition: name ":" type
type: built-in | user-defined | array | enumeration | record
user-defined: name
array: "ARRAY" "(" index {"," index} ")" "of" type
index: integer direction integer
enumeration: "(" name {name} ")"
record: "REC" "(" field {field} ")"
field: name type
direction: "to" | "downto"
built-in: "integer" | "real" | "bit" | "time" | "string" | "boolean" |
          "character"
```

CONDITIONS

A design must have been previously loaded with the **Args()** method SV/OLE must have completed the initialization of the design. (See **Args()** for a description of the initialization of the design.)

FUNCTION SVOLE.GetVariables()

INTERFACE

FUNCTION GetVariables() return String

DESCRIPTION

Returns a list of all variables and signals defined in the loaded simulation executable.

INPUTS

Boolean flag indicating if the list of signals is to be reset.

OUTPUTS

A large string containing a comma-separated list of fully prefixed VHDL identifiers. For example, "a,b,c,mux.in1,mux.in2,mux.out". For large simulation executables, this string will contain only a portion of the list of signals. To ensure that all signals are collected, the **GetVariables** method must be called repeatedly until it returns an empty string.

CONDITIONS

A simulation executable must have been previously loaded with the **Args()** method and the initialization state must be complete (see **Args()**).

EXAMPLE

```
L$ = SVOLE.GetVariables(True)
do while len(L$) <> 0
   SigList$ = SigList & L$
   L$ = SVOLE.GetVariables(False)
loop
```

```
DebugPrint “Available signals: “ & SigList$
```

FUNCTION SVOLE.GetVarType()

INTERFACE

```
FUNCTION GetVarType(String) return String
```

DESCRIPTION

Returns the type of a given variable / signal in a special format given in “outputs”, below.

INPUTS

The fully prefixed name of the variable / signal for which the type is sought.

OUTPUTS

The type as a string defined by the *type* rule of the following BNF-like grammar (see example below):

```
type: built-in | user-defined | array
user-defined: name
array: “ARRAY” “(“ index {“,” index} “)” “of” type
index: integer direction integer
direction: “to” | “downto”
built-in: “integer” | “real” | “bit” | “time” | “string” | “boolean” | “character”
```

For example, given the declaration

```
signal v : std_ulogic(7 downto 0);
```

the type of **v** returned would be the string:

```
“ARRAY (7 downto 0) of std_ulogic”
```

CONDITIONS

A simulation executable must have been previously loaded with the **Args()** method and SV/OLE must have completed the initialization of the design. (See **Args()** for a description of the initialization of the design.)

EXAMPLE

```
SigType$ = SVOLE.GetVarType(SigName$)
```

FUNCTION SVOLE.NameToAtom()

INTERFACE

```
FUNCTION NameToAtom(String) return Long
```

DESCRIPTION

Variable and signal names are each given a unique integer to represent them for more efficient internal communication and in some of the more frequently used OLE methods. This method converts a signal/variable name into its respective integer.

INPUTS

Fully prefixed name of a signal or variable.

OUTPUTS

The unique number assigned to that signal or variable.

CONDITIONS

None

EXAMPLE

```
Dim Atom as Long
Atom = NameToAtom(Signal$)
```

FUNCTION SVOLE.QueryDone()

INTERFACE

```
FUNCTION QueryDone() return Boolean
```

DESCRIPTION

Returns the partial status of the simulator. This method is retained for compatibility with other Accolade servers. **QueryStatus** is preferred.

INPUTS

None

OUTPUTS

True, if the kernel is in the **READY** state (see **QueryStatus**).

CONDITIONS

None

EXAMPLE:

```
OLEReturn = SVOLE.QueryDone()
if OLEReturn then
   UpdateTranscript("Simulation Complete!")
end if
```

FUNCTION SVOLE.QueryPercentDone()

INTERFACE

FUNCTION QueryPercentDone() return Long

DESCRIPTION

Returns a rough approximation of the percent of the current simulation that is complete.

INPUTS

None

OUTPUTS

The percent as a long integer with a range from 0 to 100.

CONDITIONS

None

EXAMPLE

```
pct = SVOLE.QueryPercentDone()
progressBar1.Value = pct
```

FUNCTION SVOLE.QueryStatus()

INTERFACE

FUNCTION QueryStatus() return Integer

DESCRIPTION

Returns the current status of SV / OLE.

INPUTS

None.

OUTPUTS

The status of SV/OLE as an integer defined as follows:

Status	Value	Description
RUNNING	0	Simulation is executing.
READY	1	Ready to run.
HALTED	2	A VHDL run-time error has occurred.

*Note: An error message can be obtained with the **GetMessage** method when in the **HALTED** state.*

ASSERTION_PENDING	3	An assertion failed and the simulator is waiting for the interface to handle the assertion (see **GetAssertion**).
INPUT_PENDING	4	Waiting for user input. This state results from a VHDL call to the **readline** procedure on the file **INPUT**. The interface must handle this state (see **SetInput**).
OUTPUT_PENDING	5	Waiting for the interface to handle user output. This state results from a VHDL call to the writeline procedure with

		the file **OUTPUT**. See **GetOutput** for information on handling this.
DEAD	6	A unrecoverable fatal error has occurred in the simulator and it must be terminated with the **Exit** method. An error message will have been reported to the transcript.

CONDITIONS

None.

EXAMPLE

```
ProcessStatus = SVOLE.QueryStatus()
Select Case ProcessStatus
        Case PROCESS_RUNNING
          . . .
        Case PROCESS_READY
          . . .
        Case PROCESS_HALTED
          . . .
End Select
```

FUNCTION SVOLE.Reset()

INTERFACE

```
FUNCTION Reset() return Boolean
```

DESCRIPTION

Reloads the simulation executable, resets simulation time back to 0, and clears all watch data. (The watches will still be active, however).

INPUTS

None.

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method and SV/OLE must be in the **READY** state (see **SVOLE.QueryStatus()**).

EXAMPLE:

```
OLEStatus = SVOLE.Reset
```

FUNCTION SVOLE.Run()

INTERFACE

```
FUNCTION Run() return Boolean
```

DESCRIPTION

Identical to the **Start()** method.

FUNCTION SVOLE.RunForever()

INTERFACE

```
FUNCTION RunForever() return Boolean
```

DESCRIPTION

Starts the simulation for the amount of time given by time'right-now.

INPUTS

None

OUTPUTS

True, if successful.

CONDITIONS

A simulation executable must have been previously loaded with the **Args()** method and SV/OLE must be in the **READY** state (see **SVOLE.QueryStatus()**).

EXAMPLE

```
OLEStatus = SVOLE.RunForever
```

FUNCTION SVOLE.SetBP()

INTERFACE

```
FUNCTION SetBP(String) return String
```

DESCRIPTION

Sets a break point for source level debugging. Since it is not possible to set a break point at any line in a file, nor is every line executable, the actually break point will be set at the closest possible line *following* the desired line. More than one break point can exist at a time, and break points are never removed, except by the **DeleteBP()** method. The actual break point returned by this method should be retained, since it is

the location of the actual break point that is required by the **DeleteBP()** method, and may differ from the specified break point location.

INPUTS

A string specifying the location of the break point. The string should consist of the module name (not the path), followed by a colon, followed by the line number. There should be no white space in the string. For example, “adder:23”.

OUTPUTS

Returns the actual break point that was set in the same form as the input.

CONDITIONS

A design must have been previously loaded with the **Args()** method and the kernel must be in the **READY** state (see **QueryStatus**).

EXAMPLE

```
dim ActualBP as string

ActualBP$ = SVOLE.SetBP(RequestedBP$)
```

FUNCTION SVOLE.SetInput()

INTERFACE

```
FUNCTION SetInput(String) return Boolean
```

DESCRIPTION

Sets the text that should be returned as the user input. This method must be called by the interface whenever SV/OLE enters the I**NPUT_PENDING** state. This state results from a user call to the IEEE standard **readline()** subprogram with the file pointer **INPUT**.

INPUTS

The text to be returned to the **readline()** call.

OUTPUTS

True, if successful.

CONDITIONS

SV/OLE must be in the **INPUT_PENDING** state.

EXAMPLE

```
If KeyAscii = 13 Then   'Enter key pressed
   If ProcessStatus = 4 Then  'INPUT_PENDING
      SVOLE.SetInput (textCommand.Text)
   Else
      ExecuteCommand (textCommand.Text)
   End If
End If
```

FUNCTION SVOLE.SingleStep()

INTERFACE

```
FUNCTION SingleStep() return Boolean
```

DESCRIPTION

Starts the simulation for one *execution step*. An execution step is either the calculation of one signal driver, or the execution of one sequential statement, in the case when the simulation is stopped within a VHDL process.

INPUTS

None.

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method and the kernel must be in the **READY** state (see **QueryStatus**).

EXAMPLE

```
   'We need to loop until we have a new line or
   'source file returned...
Do While True
   If SVOLE.SingleStep() = False Then Exit Do
   BPString$ = SVOLE.CurrentBP()
   i% = InStr(BPString$, ":")
   ModuleName$ = Left$(BPString$, i% - 1)
   LineNumber% = Val(Mid$(BPString$, i% + 1))
   'Different file or different line?
   If ModuleName$ <> LastModule$ Or LineNumber% <> LastLine Then
      Exit Do
   End If
   DoEvents    'Allow Stop button, etc. to be clicked (could get stuck here)
Loop
```

FUNCTION SVOLE.Start()

INTERFACE

FUNCTION Start() return Boolean

DESCRIPTION

Starts the simulation running. SV/OLE will run for the number of time units indicated by the time step (see the **TimeStep** property) unless interrupted (see the **Stop** method).

INPUTS

None

OUTPUTS

True, if successful.

CONDITIONS

A simulation executable must have been previously loaded with the **Args()** method and SV/OLE must be in the **READY** state (see **QueryStatus**).

EXAMPLE

```
SVOLE.TimeUnits = TimeUnitString$    ' "ns", "ps", etc.
SVOLE.TimeStep = Format$(EndTime)
OLEStatus = SVOLE.Start
```

FUNCTION SVOLE.Stop()

INTERFACE

FUNCTION Stop() return Boolean

DESCRIPTION

Halts the current simulation. SV/OLE is non-preemptive, meaning that the simulation will stop as soon as possible. "As soon as possible" means: if SV/OLE was computing the new value of a signal driver, the simulation will stop after the new value is fully computed. If SV/OLE was executing a user defined or implicit VHDL process, then it will stop as soon as that process suspends. If the **Stop()** method is called during the execution of a **readline** function from the text I/O predefined file **INPUT**, then the empty string "" will be returned from the read call and execution will be aborted as soon as the containing process suspends. Note, however, that the **Stop** method returns immediately (regardless of whether the current processing has yet halted), so the **QueryStatus()** method must be used to determine when the simulation has actually stopped.

INPUTS

None

OUTPUTS

True, if successful.

CONDITIONS

SV/OLE must be in state **RUNNING** or **INPUT_PENDING**.

EXAMPLE

```
if UserPressedStop = True then
   OLEStatus = SVOLE.Stop
```

FUNCTION SVOLE.TimeNow()

INTERFACE

FUNCTION TimeNow() return Long

DESCRIPTION

Returns the current simulation time.

INPUTS

None

OUTPUTS

The current simulation time in the units specified by the **TimeUnits** property.

CONDITIONS

The value returned by this method may be corrupted unless SV/OLE is in one of the following states (see **QueryStatus()** of the simulation server) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
Dim CurTime As Long
CurTime = SVOLE.TimeNow
```

PROPERTY SVOLE.TimeStep

INTERFACE

PROPERTY String TimeStep

DESCRIPTION

The time step is the amount of time that is simulated when the **Start** method is called. (It is not the ending simulation time.) The string should contain an integer number of time units. The time unit is taken to be the current value of the **TimeUnits** property, at the time that **TimeStep** is changed.

CONDITIONS

The **TimeUnits** property must be set before setting this property.

EXAMPLE:

```
SVOLE.TimeUnits = TimeUnitString$    ' "ns", "ps", etc.
SVOLE.TimeStep = Format$(EndTime)
OLEStatus = SVOLE.Start
```

PROPERTY SVOLE.TimeUnits

INTERFACE

PROPERTY String TimeUnits

DESCRIPTION

All time values are exchanged between the SV/OLE server and the client application without explicit time units. This property is set to the implicitly assumed time unit for all exchanged time values. It is a string containing one of the unit names of predefined type **TIME** ("fs", "ps", "ns", "us", "ms", "sec", "min").

CONDITIONS

None

EXAMPLE

```
SVOLE.TimeUnits = TimeUnitString$    ' "ns", "ps", etc.
SVOLE.TimeStep = Format$(EndTime)
OLEStatus = SVOLE.Start
```

Simulation History (SimHist) Interface

The simulation history data (encapsulated in an OLE server object called ***SimHist***) are obtained by calling the **GetSimHist()** method of the SV/OLE simulation server (**SVOLE**). The purpose of this server object is to allow the client to request from the simulator server all value changes (events) for selected signals or variables. The record of events is maintained by the SV/OLE simulation server, and may be accessed by the client whenever the server is in a **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING** state. Typical uses of this client/server relationship are to create waveform displays, perform trace dumps, or dynamically exchange simulation data with other applications such as schematic editors.

Before use, a **SimHist** object must first be created with the **GetSimHist** method (described in the previous section), as shown below (Visual Basic):

```
Set SimHist = SVOLE.GetSimHist
```

The basic order of operations when accessing simulation event data via the **SimHist** object is as follows:

1. Create the **SimHist** object using the **GetSimHist** method as described above.
2. Indicate to SV/OLE which signals/variables are of interest, using the **AddWatch()**, **DeleteWatch()** and other methods.
3. Start the simulation (as described in the previous section), allowing SV/OLE to record all value changes (events) for the selected signals or variables.

4. When SV/OLE is in the **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING** state, access the **SimHist** event data to display or otherwise act upon the event data.
5. Depending on the nature of the interface, the **ClearAll()** and **ClearEvents()** methods may be used to clear the event data in the **SimHist** object. This clearing of data can be done whenever SV/OLE is in the **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING** state. Clearing the data can help to avoid storing event data twice, and can save memory.

FUNCTION SimHist.AddWatch()

INTERFACE

FUNCTION AddWatch(String) return Boolean

DESCRIPTION

Adds a particular signal/variable to SV/OLE's list of "watched" signals and variables. These are the signals and variables for which SV/OLE is recording value changes during the simulation. A watch can be added and removed from the list at any time (almost, see conditions below). The changes in value are recorded only for the duration for which the signal or variable is on the list.

INPUTS

The fully prefixed name of the signal or variable to add to the list.

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method. SV/OLE must be in one of the following states (see **SVOLE.QueryStatus()**) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
OLEStatus = SimHist.AddWatch (SigName$)
```

FUNCTION SimHist.ClearAll()

INTERFACE

```
FUNCTION ClearAll() return Boolean
```

DESCRIPTION

Deletes all recorded values for all signals. Any changes that occur in the future will still be recorded for the signals and variables on SV/OLE's watch list. The client application may do this to free memory resources, if this information is no longer needed or has been copied by the client.

INPUTS

None

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method. The SV/OLE server must be in one of the following states (see **SVOLE.QueryStatus()**) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
'Clear all events...
OLEStatus = SimHist.ClearAll
```

FUNCTION SimHist.ClearEvents()

INTERFACE

```
FUNCTION ClearEvents(Atom: Long) return Boolean
```

DESCRIPTION

Deletes all recorded values for the given signal or variable. Any changes that occur in the future will still be recorded if the signal/variable is on SV/OLE's watch list. The client application may do this to free memory resources, if this information is no longer needed or has been copied by the client.

INPUTS

Atom is the unique integer assigned to the variable or signal whose records are to be removed (see **SVOLE.NameToAtom**).

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method and a watch must have been created for the desired signal or variable with the **AddWatch()** method. SV/OLE must be in one of the following states (see **SVOLE.QueryStatus()**) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
'Clear all events for a signal...
OLEStatus = SimHist.ClearEvents(Atom)
```

FUNCTION SimHist.DeleteEvents()

INTERFACE

```
FUNCTION DeleteEvents(Atom:Long,From:Long,To:Long) return Boolean
```

DESCRIPTION

Deletes the events recorded during a given range of simulation time. Any recorded events before and after the range will remain, and any changes that occur in the future will still be recorded, if the signal/variable is on the simulator's watch list.

INPUTS

Atom is the unique integer assigned to the variable or signal whose records are to be removed (see **SVOLE.NameToAtom()**). The **From** and **To** arguments specify the time range of the record to be deleted. The assumed time units for these time values is taken from the TimeUnits property.

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method and a watch must have been added for the desired signal/variable with the **AddWatch()** method. The kernel must be in one of the following states (see **SVOLE.QueryStatus()**) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
'Clear events for a range of time...
OLEStatus = SimHist.DeleteEvents(Atom, StartTime, EndTime)
```

FUNCTION SimHist.DeleteWatch()

INTERFACE

```
FUNCTION DeleteWatch(String) return Boolean
```

DESCRIPTION

Removes a particular signal/variable from the list of "watched" signals and variables. These are the signals and variables for which SV/OLE is recording value changes during the simulation. A watch can be added and removed from the list at any time (almost, see conditions below). The changes in value are recorded only for the duration for which the signal/variable is on the list.

INPUTS

The fully prefixed name of the signal or variable to remove from the list.

OUTPUTS

True, if successful.

CONDITIONS

A design must have been previously loaded with the **Args()** method and a watch must have been added for the given signal or variable with the **AddWatch()** method. The kernel must be in one of the following states (see **QueryStatus()** of the simulation server) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
'Delete a watch...
OLEStatus = DeleteWatch(Signal$)
```

FUNCTION SimHist.GetEvents()

INTERFACE

FUNCTION GetEvents(Atom:Long,From:Long,To:Long) return EventIterator Object

DESCRIPTION

Returns an **EventInterator** OLE object which can be used to access the recorded values of a given signal or variable record during a certain duration of time. The **EventIterator** OLE object is described later in this chapter.

INPUTS

Atom is the unique integer assigned to the variable or signal whose record that is to be accessed (see **NameToAtom** of SV/OLE). The **From** and **To** fields specify the time range of the record to be accessed. The assumed time units for these time

values is taken from the **TimeUnits** property. For example, if the atom for the signal **clk** is 1 and the **TimeUnits** property is "ns", then the call **GetEvents(1,100,200)** will return an iterator containing the value changes of the signal **clk** that occurred between 100ns and 200ns.

OUTPUTS

An EventIterator OLE object (described below).

CONDITIONS

A design must have been previously loaded with the **Args()** method and a watch must have been added for the desired signal or variable with the **AddWatch()** method. SV/OLE must be in one of the following states (see **QueryStatus()** of the simulation server) **READY**, **INPUT_PENDING, OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
Set EventIterator = SimHist.GetEvents(Atom, StartTime, EndTime)
emptyflag = SimHist.IsEmpty   'See if there are events for this signal
If emptyflag = False Then       'If there are events...
   'Traverse the list to get the events...
```

FUNCTION SimHist.GetValueAt()

INTERFACE

```
FUNCTION GetValueAt(Atom:Long,Time:Long) return Event Object
```

DESCRIPTION

Returns an **Event** OLE object, which provides access to the given signal/variable value at the given time. The **Event** OLE object is described later in this document.

INPUTS

Atom is the unique integer assigned to the variable or signal whose value is sought. **Time** is the time value for which the value is sought. The assumed time units for this time value is taken from the **TimeUnits** property.

OUTPUTS

An **Event** OLE object (described below).

CONDITIONS

A design must have been previously loaded with the **Args()** method and a watch must have been added for the desired signal or variable with the **AddWatch()** method. SV/OLE must be in one of the following states (see **QueryStatus()** of the simulation server) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
Event = SimHist.GetValueAt(Atom, T)
UpdateTranscript("Value at time " & Format$(T) & " is " & Event.Value)
```

FUNCTION SimHist.GetWatches()

INTERFACE

```
FUNCTION GetWatches() return String
```

DESCRIPTION

Returns the list of signals/variables currently being "watched" by SV/OLE. These are the signals and variables for which SV/OLE is recording changes in value during the simulation.

INPUTS

None

OUTPUTS

A string of comma separated fully prefixed variable and signal names.

CONDITIONS

A design must have been previously loaded with the **Args()** method. The kernel must be in one of the following states (see **QueryStatus()** of the simulation server) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
Dim Watches As String
Watches$ = SimHist.GetWatches
```

FUNCTION SimHist.TimeNow()

INTERFACE

```
FUNCTION TimeNow() return Long
```

DESCRIPTION

Returns the current simulation time.

INPUTS

None

OUTPUTS

The current simulation time in the units specified by the **TimeUnits** property.

CONDITIONS

The value returned by this method may be corrupted unless the kernel is in one of the following states (see **SVOLE.QueryStatus()**) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
Dim CurTime as Long
CurTime = SimHist.TimeNow
```

PROPERTY SimHist.TimeUnits

INTERFACE

```
PROPERTY String TimeUnits
```

DESCRIPTION

All time values exchanged between the server and client are without explicit time units. This property is set to the implicitly assumed time unit for all exchanged time values. It is a string containing one of the unit names of predefined type TIME (“fs”, “ps”, “ns”, “us”, “ms”, “sec”, “min”).

CONDITIONS

None

EXAMPLE

```
SimHist.TimeUnits = “ps”
```

Event Iterator (EventIterator).

The event iterator (**EventIterator** server object) is obtained only by calling the **GetEvents()** method of the OLE simulation history server. The purpose of this server object is to provide the client application access to a list of events.

An event for this purpose is defined as a change in value of a signal or variable that occurs at a certain point in simulation time. The iterator merely provides a means of traversing a list of such events.

There are two mechanisms for obtaining the events, either one-at-a-time via the **Event** object, or in "chunks" using the **GetAsString** method. The **GetAsString** method is provided to overcome speed problems that may occur do the overhead of making OLE calls. The **Event** object interator methods for accessing individual events requires a few OLE calls for each event in the list. The **GetAsString** method, on the other hand, will typically require two orders of magnitude fewer OLE calls when accessing large number of events.

During the following descriptions you should assume that there is some imaginary pointer, for each of the two methods described, which points to some location in the list of events. Note that only one of the access methods described (either **GetAsString** or the methods that return **Event** objects) should be used exclusively for a given list.

The following sections provide a reference of all of the methods available from the default dispatch of this OLE server object.

FUNCTION EventIterator.Current()

INTERFACE

FUNCTION Current() return Event Object

DESCRIPTION

Returns the **Event** object pointed to by the current pointer.

INPUTS

None

OUTPUTS

Returns the **Event** object pointed to by the current pointer.

CONDITIONS

The **IsEmpty()** method should be called first, to ensure that the list is not empty.

EXAMPLE

```
'Get the event object at the current position...
Event = EventIterator.Current
```

FUNCTION EventIterator.First()

INTERFACE

```
FUNCTION First() return Event Object
```

DESCRIPTION

Sets the current pointer for the **Event** object to the first event in the list, and returns the first event as that object.

INPUTS

None

OUTPUTS

Returns the first **Event** object in the list. If the list is empty, returns NULL.

CONDITIONS

The **IsEmpty()** method should be called first, to insure that the list is not empty.

EXAMPLE

```
'Get the first event in the list...
Event = EventIterator.First
```

FUNCTION EventIterator.GetAsString()

INTERFACE

```
FUNCTION GetAsString() return String
```

DESCRIPTION

Returns some number of events (a maximum of 200), of the events in the list, starting with the event pointed to by the current pointer of the string method. The current pointer is updated to point to the event following the last one returned by this call. Each event is represented as text, and all the returned events are returned in a single string. The method can be called repeatedly to obtain all the events in the list.

INPUTS

None

OUTPUTS

A large string of event text. The first part of the string contains a character followed by an integer number and indicates the status of the traversal of the list. The three possible values and their meaning follow:

Character	**Description**
E (empty)	The list is empty; the integer number will be 0.
C (complete)	There are no events in the list after the ones being returned by this call. The integer number is the number of events returned by this call.
I (incomplete)	There are still events in the list after the ones being returned by this call. The integer number is the number of events returned by this call.

The character is followed by a single space, and the integer number is right justified in a field of 10 characters. The integer is immediately followed by a colon. Following the colon is a list of colon separated events. Each event is represent as an integer time value followed by a single space and the text representation of the new value stored at that time value. The time value assumes the implicit time unit specified by the **TimeUnits** property of the **SimHist** OLE server object. The text representation of values is described for the **Value()** method of the **Event** object later in this chapter. An example of a string returned by this method for a bit signal might be: "C 3:0 0: 10 1: 20 0", meaning the signal became 0 at time 0, 1 at time 10 and 0 again at time 20.

CONDITIONS

The kernel must be in one of the following states (see **SVOLE.QueryStatus()**) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or **ASSERTION_PENDING**.

EXAMPLE

```
EventString$ = EventIterator.GetAsString   'Get a batch of events
EventCode$ = Left$(EventString$, 1)
Do While EventCode$ <> "E"
   e% = 1                          'Current offset into the EventString
   ' Now read the Events...
   Do While True
      'Find next event in the string...
      e% = InStr(e%, EventString$, ":")
      If e% = 0 Then
         Exit Do              'Ran out of events
      Else
         e% = e% + 1
      End If
      'e% now points to the first event time
      e1% = InStr(e%, EventString$, " ")
      If e1% <= e% Then Exit Do
      'Parse out the event time...
      EventTimeString$ = Mid$(EventString$, e%, e1% - e%)
      EventTime = Val(ET$)
      e1% = e1% + 1            'Move to start of data value field
      'Get the event data field...
      e2% = InStr(e1%, EventString$, ":") 'Find end of data field
      If e2% <= e1% Then
         e2% = Len(EventString$) + 1
      Else
         e2% = e2%
      End If
      EventValue$ = Mid$(EventString$, e1%, e2% - e1%)

      '****Process the event (display, etc.) here****

   Loop    'Events in the object
   If EventCode$ = "C" Then    'We are done (complete list)
      Exit Do
   End If
   EventString$ = EventIterator.GetAsString       'Get next batch of events
   EventCode$ = Left$(EventString$, 1)
```

```
Loop    'GetEventAsString loop
```

FUNCTION EventIterator.IsEmpty()

INTERFACE

FUNCTION IsEmpty() return Boolean

DESCRIPTION

Determines if the **EventIterator** list is empty.

INPUTS

None

OUTPUTS

True, if the list is empty.

CONDITIONS

None

EXAMPLE

```
emptyflag = EventIterator.IsEmpty   'See if there are events for this signal
If emptyflag = False Then       'If there are events...
   'Process the events...
```

FUNCTION EventIterator.IsFirst() return Boolean

INTERFACE

FUNCTION IsFirst() return Boolean

DESCRIPTION

Determines, if the current pointer of the **EventIterator** list is pointing to the first element in the list.

INPUTS

None

OUTPUTS

True, if the current pointer points to the first event in the list.

CONDITIONS

The **IsEmpty()** method should be called first, to ensure that the list is not empty.

EXAMPLE

```
firstflag = EventIterator.IsFirst
```

FUNCTION EventIterator.IsLast()

INTERFACE

```
FUNCTION IsLast() return Boolean
```

DESCRIPTION

Determines, if the current pointer of the **EventIterator** list is pointing to the last element in the list.

INPUTS

None

OUTPUTS

True, if the current pointer points to the last Event in the list.

CONDITIONS

The **IsEmpty()** method should be called first, to ensure that the list is not empty.

EXAMPLE

```
lastflag = EventIterator.IsLast
```

FUNCTION EventIterator.Last()

INTERFACE

```
FUNCTION Last() return Event Object
```

DESCRIPTION

Sets the current pointer for the **EventIterator** list to the last event in the list, and returns the last event as an object.

INPUTS

None

OUTPUTS

Returns the last **Event** object in the list, if the list is not empty, otherwise NULL.

CONDITIONS

The **IsEmpty()** method should be called first, to insure that the list is not empty.

EXAMPLE

```
Event = EventIterator.Last
```

FUNCTION EventIterator.Next()

INTERFACE

```
FUNCTION Next() return Event Object
```

DESCRIPTION

Moves the current pointer for the **EventIterator** list to the next event in the list, and returns the next event as an object.

INPUTS

None

OUTPUTS

Returns the next **Event** object in the list, if the current pointer was not previously at the end of the list.

CONDITIONS

The **IsLast()** method should be called first, to ensure that the pointer is not currently at the end of the list.

EXAMPLE

```
Event = EventIterator.Next
```

FUNCTION EventIterator.Previous()

INTERFACE

FUNCTION Previous() return Event Object

DESCRIPTION

Moves the current pointer for the **EventIterator** list to the previous event in the list, and returns the previous event as an object.

INPUTS

None

OUTPUTS

Returns the previous **Event** object in the list, if the current pointer was not previously at the beginning of the list.

CONDITIONS

The **IsFirst()** method should be called first, to insure that the pointer is not current at the beginning of the list.

EXAMPLE

Event = EventIterator.Previous

FUNCTION EventIterator.Reset()

INTERFACE

FUNCTION Reset() return void

DESCRIPTION

Resets the current pointer for the **GetAsString** method back to the beginning of the list.

INPUTS

None

OUTPUTS

None

CONDITIONS

None.

EXAMPLE

```
EventIterator.Reset
```

Event Object (Event).

The event server object (**Event**) is obtained only by calling certain methods of the **SimHist** and **EventIterator** server objects. The **Event** server object provides access to the data of a specific event. The following sections describe all of the methods available from the default dispatch of this OLE server.

FUNCTION Event.Time()

INTERFACE

```
FUNCTION Time() return Long
```

DESCRIPTION

Returns the time at which this event occurred.

INPUTS

None

OUTPUTS

Integer time value. The implicit time unit is that of the **TimeUnits** property of the **SimHist** OLE server.

CONDITIONS

None

EXAMPLE

```
Event = EventIterator.Next
UpdateTranscript("Value is " & Event.Value & " at time " &
Format$(Event.Time)
```

FUNCTION Event.Value()

INTERFACE

```
FUNCTION Value() return String
```

DESCRIPTION

Returns the value of the signal or variable at this event.

INPUTS

None

OUTPUTS

A string representation of the value. If the value is a scalar value it is represented as follows:

Type	**Representation**
Integer	The integer value.
Bit	The integer 0 for '0' and 1 for '1'.
Boolean	The integer 0 for FALSE, 1 for TRUE.
Enumeration	The integer representation of the enumerate. Each enumerate is assigned a successive integer number as declared in the enumeration declaration from left to right, starting with 0. Thus, the integer for (RED,WHITE,BLUE) would be 0 for RED, 1 for WHITE, and 2 for BLUE.
Character	The character enclosed by single quotes.
Real	The real value.
Time	An integer specifying the time in the base unit of the type Time.
String	The string.

If the type of the value is an array or a record, than its text representation is a comma separated list of the text representations of its elements. For arrays the list is given from the 'left element to the 'right element. For records, it is given as they appear in the record definition from the first element to the last element.

CONDITIONS

The kernel must be in one of the following states (see **QueryStatus()** of the simulation server) **READY**, **INPUT_PENDING**, **OUTPUT_PENDING**, or

ASSERTION_PENDING.

EXAMPLE

```
Event = EventIterator.Next
UpdateTranscript("Value is " & Event.Value & " at time " &
Format$(Event.Time)
```

Index

C

D

E

S

T

U

V

W

Z