Using Package Signals for Fun and Profit

Michael D. McKinney
Sr. ASIC Design Engineer
PCIBus Solutions Group
Texas Instruments, Inc.
Sherman, TX 75090
Phone: 903-868-7202
email: mmckinney@shvp.sc.ti.com

Abstract

As ASIC designs become larger and more complex, the functional verification environments (the testbenches) must become more comprehensive in scope if they are to provide the basis for adequate validation and verification of the designs. This does not mean, however, that the testbench must itself become more complex or that it becomes harder to use. This paper presents details for using VHDL Signals in a unique way to simplify and ease communication between different components of a testbench, and provide versatility to users of the environment when generating test vectors.

Definition: Testbench

In order to adequately define the scope of this VHDL technique it is necessary to state clearly, at least for the purposes of this paper, the definition of a ‘testbench’. A testbench is a self-contained Entity / Architecture pair, generally written in structural or netlist style. The words ‘self-contained’ indicate two key aspects of the testbench. The first aspect is that the Entity of the testbench has no Port statement, and means that the testbench itself cannot be affected by any outside influence. The second, related, aspect is that the testbench Architecture contains Components or Processes which represent the three required activities for functional simulation: 1) some sort of stimulus activity (Component, Process, file reader, command parser, etc.), 2) a Unit Under Test (UUT), generally the component, ASIC or system being designed and verified, and which receives the stimulus and provides responses, and 3) some sort of response checker mechanism (Figure 1).

The UUT is normally the result of a design team’s effort and is usually written in the RTL style of VHDL for use with synthesis tools. In contrast, the remaining testbench components are written in the behavioral or algorithmic style of VHDL. This style takes full advantage of the power of the language to affect the overall efficiency and speed of the simulator environment and its use of machine core space. In turn this focus on testbench efficiency affects the productivity of the person(s) responsible for performing the simulations which produce the test vectors for the UUT.
The VHDL technique discussed in this paper (i.e., the use of Signals defined in Packages) introduces a powerful and useful method to create communication between testbench components as necessary without sacrificing testbench ease-of-use or test designer productivity.

![Diagram showing Stimulus Generator, UUT, and Response Checker as self-contained components.]

**Figure 1**

---

**Signals, Packages, and Package Signals**

Typically, a Signal is declared in an Architecture and is known and able to be evaluated and driven without redeclaration within any Process, Block, Procedure or Function in that Architecture. The scope of such signals does not extend into any Component defined by the Architecture, nor to any VHDL Entity outside the Architecture at all. A signal's value may be passed to any other Entity only through connection with a Port Signal of that Entity. By usage and convention the typical place for the declaration of Signals in a VHDL design is in an Architecture.

The most common use of Packages in VHDL environments is to declare Constants, Procedures, Functions and even Components that are or may be useful to all of the Architectures used in a particular simulation. The Packages themselves are treated much like an 'include' file would be treated in application languages such as C or C++. In this way any object declared in a Package has the potential to become global in scope, by the inclusion of Library and Use statements in the VHDL source of any Entity or Architecture which needs to make use of that object. It is important to emphasize that such objects are *potentially* global, but in practice never are. The contents of a Package are typically made visible to a select subset of Architectures and ignored by the rest (Figure 2).

Signals that are declared in Packages are identical in basic nature to any other Signal anywhere in the environment: they may have several drivers, they may be resolved, they do not change value until the Process driving them is suspended, any change in their value forces another delta time iteration in the simulator, etc. Their
only claim to fame is that, by their inclusion in a Package, they potentially may be evaluated and/or driven by every concurrent or sequential Process in every Architecture in the entire design -- which in most cases is a testbench and includes all Architectures and Processes of the UUT.

Disclaimer

The use of Signals declared in Packages is a valuable and powerful VHDL testbench technique which can solve some tough problems. It is not, however, a synthesis technique. This paper does not encourage the use of Package Signals in any set of VHDL source code which is destined for synthesis. In fact, it is not known how any of the synthesis engines may use such Signals, since they are not declared in an Architecture or Entity. This paper furthermore does not encourage the frivolous use of this technique without careful study of the needs of the testbench and of the design team. Without the full understanding of the potential of this technique, the use of Package Signals without constraints could create a lot of heartache and confusion for users of a testbench in simulation as well as the maintainers of its source code.

Being so forewarned, the remainder of this paper will describe the proper use of Package signals, indicate some of the pitfalls that users need to watch out for, and show an example of the type of problem that is best served by the use of Package Signals.
Package Signal "Gotchas"

Syntactically, making use of Package Signals in a VHDL simulation is simple enough: 1) analyze the package into some library; 2) make the Signals declared there visible to the appropriate Architectures by including Library and Use statements in their source files; 3) analyze the Architectures. Now that the Signal is actually in place, however, there are four important concepts to keep in mind which determine how the Signal actually performs, how the Signal affects other components in the design, and how use of the methodology affects the design group.

* 1. Documentation, documentation, documentation ...

First (it almost goes without saying ...) is to fully document any effort with Package Signals. This is most important in order to protect not only the investment in time, but also to protect the methodology from the new and less experienced engineer at some future time. They will look for Signals (at least the ones being driven) to be defined in Architectures. Without adequate documentation indicating where the Signals are declared and what Architectures know about them, significant confusion may be generated. "How in the world can a signal be successfully driven in an Architecture without being defined there? The code analyzes without error, but a 'grep' for the Signal name produces only the line containing the assignment ... What? How?" Therefore, good documentation should be a pretty strict requirement.

* 2. Communication Between Architectures Bypass Entities ...

Second is to recognize that this methodology creates communication between Architectures regardless of their position in the design hierarchy, and without the use
of Port Signals (Figure 3). With this methodology the behavior of an Architecture can affect the behavior or results within a completely unrelated Block, Process, Component or Architecture. Even being so potentially confusing, however, it is this aspect of the methodology which makes it so useful in a testbench. It is this aspect which typically drives the decision to utilize Package Signals in the first place.

* 3. Multiple Instances of Architectures Using Package Signals ...

Third, a most gnarly problem, is the multiple instantiation of an Architecture which is driving or evaluating Package Signals. Up to this point this has not been an issue, because the only Signals known to the Architecture are those declared inside the Architecture. However, as previously noted, Package Signals operate in exactly the same way as Signals declared in any other section of the design! This means that if two Architectures (or any Process, Block or Component within) are driving that signal, there may be potential conflict. The typical answer to this scenario when there is conflict is to either rename the signal being driven in one of the Architectures, or modify the behavior of the Architecture (one or both) to remove the conflict.

But what is the answer when the two Architectures are instances of the same source code, the same VHDL model? Neither remedy previously mentioned applies, since any change to the source of the Architecture or in the name of the Package Signal is immediately reflected in both instances, and the conflict remains in place. The answer is in three parts, one applied to the Package Signal and two applied the Architecture itself (Figure 4).

Part 1 is to change the Package Signal from a single unit of some type into an array of those types whose index relates to the number of projected instances to be used. For example, if the Package Signal had been declared as

\[
\text{Signal xyz : std\_logic := 'Z'};
\]

it would be changed to

\[
\text{Signal xyz : std\_logic\_vector(0 to 9) := (others => 'Z');}
\]

If the Package Signal had been declared as an unresolvable type, such as

\[
\text{Signal abc : boolean := false;}
\]

it becomes necessary for the Package to declare a new type which is an array of the unresolvable type, which is then used as the new declaration of the Signal:

\[
\text{Type boolvec is array(0 to 9) of boolean;}
\]

\[
\text{Signal abc : boolvec := (others => false);}
\]

Part 2 is to send the Architecture a new Generic value through the Generic Map clause of the Entity to which it is attached. This Generic will indicate the "instance position" of the Architecture in the structure of the design. At compile time this Generic value can be set to a default value:
Entity Entity_Name is
  Generic ( instance : natural range 0 to 9 := 0 );
  Port ( ***** );
End Entity_Name;

Library Globals;
UseGlobals.Package.Package_Signal; -- or .All, as required ...

Architecture Architecture_Name of Entity_Name is
  *****
End Architecture_Name;

and at some later time can be changed to a more reasonable value by a Configuration
Specification or Configuration Declaration:

For label : component_name
  use entity Library_Name.Entity_Name(Architecture_Name)
  generic map (instance => 1);
end for;

--- Figure 4 ---

Part 3 completes the sequence by modifying the Architecture source at the
line(s) which drive or sense the Package Signal. Rather than driving or sensing the
original Signal:
xyz <= '1'; OR if (abc) then ***

the source would now drive or sense the “instance” element of the new Signal:

xyz(instance) <= '1'; OR if (abc(instance)) then ***

At this point any potential conflict in these Package Signals has been removed, since each Architecture, whatever its “instance position”, is driving only one array element of the single Package Signal.

* 4. Resolvable and Unresolvable Signal Types

This aspect of using this methodology is one which involves the choice of Type for the Package Signal. If the communication between the Architectures is to be one-way, then the type chosen should be an unresolvable type. ‘One-way’ communication would indicate that the Package Signal is always driven or always received by a single Architecture, but never both. Unresolvable types would include such types as integer, boolean, bit, X01 and std_ulogic. This is a protective choice because if it turns out that two Architectures are in fact driving the Package Signal, the analysis will be unsuccessful.

If however, the communication needs to be two-way, then a resolved type such as std_logic may be chosen. Be aware, however, that if a resolved type is chosen, no analysis error will occur, even though there may in fact be a functional conflict on that Signal during simulation.

One further issue needs to be addressed on resolvable and unresolvable Signal types which focuses on the std_ulogic base type as defined in IEEE Std. 1164-1993. Any Signal or Variable object declared as type std_ulogic (the unresolvable type) can be freely assigned from an object of type std_logic (the resolvable type), and vice versa. However, an object of type std_ulogic_vector (the unresolvable vector type) cannot directly accept values from an object of type std_logic_vector (the resolvable vector type), nor can the opposite function occur. The word ‘directly’ is used here, because there are functions built into IEEE Std. 1164-1993 which perform the type conversion, and in some cases a simple cast of the value is all that is required.

However, this characteristic of the std_ulogic base type will color the decision about Package Signal types when the Package Signals are required to perform only one-way communication and it is desired to have unresolvable types there to detect multiple drivers of the Package Signals at compile time.

The Example

This section will detail the use of Package Signals in a testbench used to verify a real ASIC design. Package Signals were chosen for this project because other methods had been tried and found to be not successful, too slow, or used up too much CPU resources.

* 1. The Requirements
For this project the requirement was for a VHDL Entity / Architecture which would be instanced as a Component in the testbench netlist. The function of the module would be to write a formatted file with data which would later be interpreted as TI's Test Description Language (TDL). TDL is a vector-based language which contains both stimulus and UUT response in each vector and is intended for testers which are used to test ASICs. The VHDL module would write a vector into the file for each clock cycle of simulation time. The module is therefore called a TDL Writer (unique description, isn't it?).

In the testbench netlist the Ports of the TDL Writer are declared as mode IN, and are equal in number and connected to the Ports of the UUT. The TDL Writer therefore listens passively to the signal events occurring on the Ports of the UUT. At specific times during each clock cycle of simulation, the TDL Writer senses the values appearing on each of the UUT Ports and writes out a vector of information to the output file.

* 2. TDL Writer Basic Activity

TDL specifies a unique set of characters to be used for IN ports and a second unique set of characters for OUT ports (both specified from the perspective of the UUT). The TDL Writer's interpretation of the required output character for a particular IN or OUT port in any particular vector was straightforward -- a simple type conversion from std_logic to character was performed. In VHDL, this can be expressed most simply in declarations such as these:

Type TABLE1 is array(std_logic) of character;

Constant std2input : TABLE1 := ('Y', 'Y', 'L', 'H', 'S', 'Y', 'L', 'H', 'Y');

Constant std2output : TABLE1 := ('A', 'A', '0', 'I', 'Z', 'A', '0', 'I', 'M');

followed by use of the declarations in statements such as this (assuming usage of textio):

write(outline, std2input(abc)); -- abc is the name of a
-- UUT Port of mode IN ...

write(outline, std2output(abc)); -- abc is the name of a
-- UUT Port of mode OUT ...

* 3. The INOUT Port Problem (there always seems to be one ...)

This usage is perfect for use with UUT Ports that remain in mode IN or OUT throughout the simulation. There remains, however, the problem of handling the Ports of the UUT which are declared as mode INOUT, and change their mode from time-to-time during the simulation. This is a large problem because this is the preferred mode for most of the UUT pins in actual practice.

Resolution of this problem requires one further bit of information to be passed
to the TDL Writer indicating which of the UUT Ports of mode INOUT are IN and which are OUT for the current vector. This information is commonly known within the UUT IO ring, and is expressed as the enable signal for the Port's output driver (typically a tri-state driver). When in the active state, the enabling signal turns on the output driver and the Port can be considered OUT for that vector. When inactive, the output driver is off and the Port can be considered IN for that vector.

The most common method for getting this enable information to the TDL Writer is via additional Port signals in the UUT and TDL Writer. In this case, however, there was an additional, and common, constraint: the UUT was being synthesized. This meant that any additional Ports which did not really exist in the hardware would translate into gate-level ports which would then have to be edited out. Extra and unnecessary work for everyone involved. A second method involves the use of commands to the simulator environment itself. These commands monitor the enable Signal in the UUT, and assign a translated value to a Signal or Variable within the TDL Writer. This method works, and has provided good output for a long time -- but may become too cumbersome for two reasons. First, this method is simulator environment (and therefore EDA vendor) specific. Any desire to change to another EDA vendor immediately means that the control file for the commands has to be completely rewritten. Second, when at some time a large number of commands have to be active at the same time (one command is required for each IO Port to be monitored), simulation times suffer massively simply because of the drain on CPU resources.

* 4. The INOUT Port Solution

The solution for this example was in using Package Signals (how did you know?). The first step is to write a Package of Signals which will carry the Port enable information to the TDL Writer. One Signal is declared in the Package for each required UUT Port enable signal (e.g., only for those Ports defined as mode INOUT).

The second step was to modify the UUT to drive the Package Signals with
enable information. The Architecture chosen inside the UUT was the one in which the Port enable signals are also known. The modification consisted A) the addition of the Library and Use statements as discussed earlier, and B) creation of a special, non-synthesizable Process which contained all the local enable signals in its sensitivity list and assigned each enable signal onto its Package Signal equivalent. (Note: the new Process should be non-synthesizable because the assignment statements contained therein should not be allowed to be translated to buffers in the gate level netlist. Each synthesis tool has some method for accomplishing this task, typically a synthesis tool keyword expressed as a VHDL comment line.)

The third and last step was the inclusion of an If statement in the TDL Writer for each UUT INOUT Port. This If statement evaluates a Package Signal's value (remember, it now carries the enable information for a specific Port), and uses either the std2output conversion or std2input conversion for that Port for that vector. In this manner, all Ports of the UUT are correctly evaluated for each vector being written to the output file (Figure 5). The Port enable information is passed from UUT to TDL Writer quickly and efficiently (a few delta times) without the use of new and unnecessary Ports in either UUT or TDL Writer and without the use of vendor specific environment commands. Addition of new INOUT Ports in the UUT require a new Package Signal, a new assignment in the UUT special Process, a new If statement in the TDL Writer and a new 'make' of the testbench.

Conclusion

This paper has presented a new and unique usage of VHDL Signals by declaring them in Packages. The advantages and disadvantages of this technique were discussed, and a real-world example was given for the methodology. The technique has proven to be a powerful testbench tool, and has allowed more flexibility in the use of the testbench to generate test vectors. It is hoped that the discussion will enhance the creative use of VHDL for solving real-world problems.

References


