Designing Script Driven Testbenches

Kevin A. Sholander
Senior Design Engineer
Philips Semiconductors
Albuquerque, NM 87113
sholander@abqhp1.scs.philips.com

Graciela B. Sholander
B.S. Electrical Engineering
University of California, San Diego
La Jolla, California

ABSTRACT

This paper presents a package for developing testbenches that are easy to write, are easy to maintain and automatically check the results against the expectations. These testbenches not only allow for the separation of the test patterns and the automatic checking from the VHDL but also simplify the process of generating the VHDL code so that a fairly complex testbench can be written in a matter of minutes.

The package being presented incorporates procedures and functions that will read symbolic commands from a pattern file and parse them into a set of vectors and checks for the design being tested. The testbenches that can be created with the package are suitable for testing any design that can be fully verified through its ports. This paper goes through the development of all of the procedures and functions in the package and also presents an example design and the VHDL code and stimulus file that could be used to fully verify the design.

INTRODUCTION

Now that you have completed the most complicated and intricate design that you have ever written in VHDL, you face what may be an even greater challenge -- that of verifying that the design functions according to its specification. The job of creating a testbench that will fully check a design may be a more difficult task than the original design itself.

A good testbench has several characteristics. One is that the VHDL code running the testbench must be separate from the vectors that are to be run. This way, if ever there is a need to change the vectors, such as to keep the test up to date with the design as it changes or to add tests to check for errors that were found in some other way, it can be done with minimum risk and effort. Another characteristic of a good testbench is that it will not only apply the vectors to the device, but will also test the actual results to see if they match the expected results. This greatly reduces the amount of time required to verify a design. One final characteristic of a good

testbench is that it be fairly simple and easy to create and modify. It should not take longer to write a testbench than it took to create the design in the first place.

One way to accomplish all of these seemingly contradictory goals is to develop a testbench that reads the vectors from a command file and applies them to the device under test. This allows the vectors to be freely changed and run without having to recompile or rewrite any VHDL code. The test vectors can have expected results embedded to verify that the design is behaving as intended. Achieving the last goal is probably the most difficult, since creating VHDL code that can read and parse a vector file and test the expected results can be quite an endeavor. This could be why many designers choose to proceed along a different path, ending up with a less than optimal testbench which is difficult to run and maintain but that may have saved some time in writing.

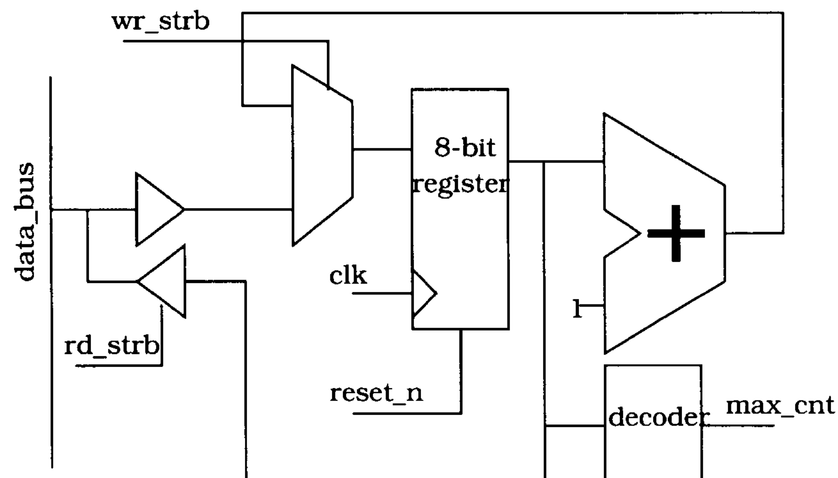
At Philips Semiconductors, we are using an internally developed package which allows us to very quickly and easily develop testbenches that meet all of the above criteria. The package takes care of all of the file I/O, string manipulation and conversions and allows for the creation of commands and variables that can be used in the testbench's vector file. The vector files that are created read almost like assembly code, with conditional statements that can check the state of the design under test or the state of the testbench itself, and can take branches or loop based on these conditions. These vector files, when properly documented, are very easily maintained if ever the test needs to be modified or expanded.

The functions and procedures in this package allow the user who is writing the package to define his or her own mnemonics as well as the actions that will be performed when these instructions are executed during the simulation. For instance, if the design to be tested is a bus interface, then some useful instructions may be read, write, and read-modify-write. The actions that should be performed for these instructions is then defined in the testbench along with the parameters that will be accepted for each of these instructions.

CREATING A SAMPLE TESTBENCH

As a very simple example, consider the design shown in figure 1. This design implements a counter whose value is readable and writable through an 8 bit data bus, with a flag that gets set when the counter has reached its maximum value. The VHDL code that implements this design could be written as follows:

Figure 1: Example counter design



```
-----------------------------------------------------------------------------------------------------
ENTITY model_under_test IS
PORT (
   clk : IN std_ulogic; -- clock signal
   reset_n : IN std_ulogic; -- active low reset
   data_bus : IN std_logic_vector(7 DOWNTO 0);
   wr_strb : IN std_ulogic; -- write strobe
   rd_strb : IN std_ulogic; -- read strobe
   max_cnt : OUT std_ulogic; -- counter flag )
END model_under_test;
ARCHITECTURE behavioral OF model_under_test IS
   SIGNAL count_value : std_ulogic_vector(7 DOWNTO 0);
BEGIN
counter : PROCESS
   BEGIN
         WAIT UNTIL clk'EVENT and clk = '1';
         IF reset_n = '0' THEN
               count_value <= "00000000";
         ELSIF wr_srb = '1' THEN
               count_value <= To_StdULogicVector(data_bus);
         ELSE
               count_value <= count_value + '1';
         END IF;
   END PROCESS;
data_bus <= To_StdLogicVector(count_value) WHEN rd_strb = '1'
               ELSE "ZZZZZZZZ";
max_cnt <= '1' WHEN count_value = "11111111" ELSE '0';
END;
-----------------------------------------------------------------------------------------------------
```

The first step is to determine which ports in the model under test need to have stimulus applied to them and which ports need to be sampled in order to fully test the model. Those ports that need to have stimulus applied will appear as BUFFER, OUT or INOUT ports in the testbench. Those ports that need to be sampled will appear as IN or INOUT ports in the testbench. Only those ports that will be both driven and sampled by the testbench should be of mode INOUT. Below is the entity for the testbench to test the sample code:

```
-----------------------------------------------------------------------------------------------------
ENTITY test_bench IS
   PORT (
         clk : BUFFER std_ulogic;
         reset_n : BUFFER std_ulogic;
         data_bus : INOUT std_ulogic_vector(7 DOWNTO 0);
         wr_strb : BUFFER std_ulogic;
         rd_strb : BUFFER std_ulogic;
         max_cnt : IN std_ulogic );
END test_bench;
-----------------------------------------------------------------------------------------------------
```

After the entity has been defined, the next step is to create the architecture that will drive all of the entity's output ports and test its input ports. No signals need to be declared in the architecture, but it is necessary to define the input file that will contain all of the stimulus commands. This is done as shown below:

```
-----------------------------------------------------------------------------------------------------
ARCHITECTURE behavioral OF test_bench IS
   FILE fptr : TEXT IS IN "stimulus.file" ;
BEGIN

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

The architecture will consist of two (or more) processes. Every free-running signal (such as clock inputs) will have a process in which that signal is driven. In the example above, there is only one free-running signal, which is clk. An example clock process is shown below:

```
-----------------------------------------------------------------------------------------------------
CLOCK_DRIVER : PROCESS
   BEGIN
         clk <= '0';
         WAIT FOR 25 ns;
         clk <= '1';
         WAIT FOR 25 ns;
END PROCESS;

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

All of the signals that are not free-running will be under the control of the second process. This second process is the one that will make use of all of the file and parse utilities from the package. Before coding the second process, it is first required to declare the necessary variables for storing the instruction set and variables that will be used in the vector file. These variables will not vary from one test bench to the next, but will remain roughly the same for all test-benches. The variables that are required to make full use of the package are declared below:

```
-----------------------------------------------------------------------------------------------------
VECTORS : PROCESS
-- file_input stores all of the lines in the file
   VARIABLE file_input : str_array(1 TO max_lines_in_file);
-- defined_vars stores all defined variables and their values
   VARIABLE defined_vars : var_struct;
-- instruction_set stores all defined instructions
   VARIABLE instruction_set : all_instructions;
-- lines_in_file stores the number of lines that were read
   VARIABLE lines_in_file : integer;
-- line_num stores the number of the current line
   VARIABLE line_num : integer;
-- command is the instruction string read from current line
   VARIABLE command : string(1 TO max_field_width);
-- The fields are used to store the parameters for each command
   VARIABLE field_two, field_three, field_four : integer;
-- accumulator is used to hold read data for future testing
   VARIABLE accumulator : integer;
-- loop_num is the current depth of looping
   VARIABLE loop_num : integer;
-- curr_loop_count stores the loop index for all loops
   VARIABLE curr_loop_count : int_array(1 TO max_nested_loop);
 -- term_loop_count stores the terminal count for all loops
   VARIABLE term_loop_count : int_array(1 TO max_nested_loop);
-- loop_line stores the starting line of each loop
   VARIABLE loop_line : int_array(1 TO max_nested_loop);
```

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

After all of the required variables have been declared, the process should begin by initializing all of the BUFFER, OUT and INOUT ports that were declared in the entity as well as initializing the variables that will be used by the package. An example initialization is shown below:

```
----------------------------------------------------------------------------------------------------
BEGIN
   -- initialize output and bidirectional ports
   reset_n <= '1';
   rd_strb <= '0';
   wr_strb <= '0';
   data_bus <= "ZZZZZZZZ";
   -- initialize package variables
   defined_vars.num_of_vars := 0;
   loop_num := 0;
   line_number := 0;

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

Once all of the variables have been declared and initialized, the next step is to define the instructions that will be used in the stimulus file to test the model. These instructions can be used to drive or sample any of the declared ports, control the flow of execution of the vector script, assert messages about the simulation, or perform just about any other function that may be wanted. A simple set of instruction definitions that could be used in the testbench for the sample design might appear as follows:

```
----------------------------------------------------------------------------------------------------
   Define_Instruction(instruction_set,"--",1);
   Define_Instruction(instruction_set,"RESET",1);
   Define_Instruction(instruction_set,"NOOP",1);
   Define_Instruction(instruction_set,"LOAD",2);
   Define_Instruction(instruction_set,"READ",1);
   Define_Instruction(instruction_set,"VERIFY",3);
   Define_Instruction(instruction_set,"BR_SET",2);
   Define_Instruction(instruction_set,"HALT",1);
   Define_Instruction(instruction_set,"LOOP",2);
   Define_Instruction(instruction_set,"END_LOOP",1);
   Define_Instruction(instruction_set,"ASSERT_DONE",1);

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

Now that all of the instructions have been defined, the script file can be read into the simulation using the Read_File procedure that is defined in the package as follows:

```
----------------------------------------------------------------------------------------------------
   Read_File(defined_vars,fptr,file_input,lines_in_file);

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

This will store the entire contents of the file (minus the comments and empty lines) into the file_input variable, and record the number of lines that were stored into the lines_in_file variable. The remainder of the VHDL testbench is a loop that goes through each line of the command file that was read and performs the desired actions. This loop is created as follows:

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

```
    WHILE (line_number < lines_in_file) LOOP
          line_number := line_number + 1;
          Parse (defined_vars, instruction_set,
                file_input(line_number)(file_input(line_number)'RANGE),
                command, field_two, field_three, field_four);
```

(Note that it was necessary to provide a range for the string pointed to by file_input(line_-number) in order to dereference the pointer into a string. Without providing this range you will end up with a compile time error indicating that there is a type mismatch. This is due to the fact that there is no way in VHDL to explicitly dereference pointer variables.)

The final step that is needed in setting up the test bench is to define the actions that will be taken for each of the commands that have been defined. This is done using a nested IF statement. A CASE statement could be used, but this would require that all of the instruction mnemonics have the same number of characters in their strings since it is not possible to overload the compare operator used in CASE statements. The "=" operator in the IF statements below has been overloaded to allow for comparisons of unequal length strings.

```
          IF (command = "--") THEN
                NULL;
          ELSIF (command = "RESET") THEN
                reset_n <= '0';
                WAIT FOR 1000 ns;
                reset_n <= '1';
          ELSIF (command = "NOOP") THEN
                WAIT UNTIL clk'EVENT and clk = '1';
          ELSIF (command = "LOAD") THEN
                WAIT UNTIL clk'EVENT and clk = '1';
                data_bus <= To_StdLogicVector(field_two,8,Unsigned);
                wr_strb <= '1';
                WAIT UNTIL clk'EVENT and clk = '1';
                data_bus <= "ZZZZZZZZ";
                wr_strb <= '0';
          ELSIF (command = "READ") THEN
                WAIT UNTIL clk'EVENT and clk = '1';
                rd_strb <= '1';
                WAIT UNTIL clk'EVENT and clk = '1';
                accumulator <= To_Integer(data_bus,Unsigned);
                rd_strb <= '0';
          ELSIF (command = "VERIFY") THEN
                IF (accumulator /= field_two) THEN
                      line_number := field_three;
                END IF;
          ELSIF (command = "BR_SET") THEN
                IF (max_cnt = '1') THEN
                      line_number := field_two;
                END_IF;
          ELSIF (command = "HALT") THEN
                ASSERT (FALSE)
                REPORT ("Simulation halted due to HALT command")
                SEVERITY note;
```

```
            WAIT;
      ELSIF (command = "LOOP") THEN
            loop_num := loop_num + 1;
            loop_line(loop_num) := line_number + 1;
            curr_loop_count(loop_num) := 0;
            term_loop_count(loop_num) := field_two;
      ELSIF (command = "END_LOOP") THEN
            curr_loop_count(loop_num) :=
                  curr_loop_count(loop_num) + 1;
            IF (curr_loop_count(loop_num) = term_loop_count(loop_num))
            THEN
                  loop_num := loop_num - 1;
            ELSE
                  line_number := loop_line(loop_num);
            END IF;
      ELSIF (command = "ASSERT_DONE") THEN
            ASSERT (FALSE)
            REPORT "Simulation completed without errors"
            SEVERITY note;
      ELSE
            ASSERT (FALSE)
            REPORT "Illegal instruction in input file"
            SEVERITY error;
            WAIT;
      END IF;
   END LOOP;
```

This loop will be exited when the last command from the file has been executed. The last lines of the process should report that the simulation is complete, then just wait to stop any other stimulus from being generated. This code would appear as follows:

```
   ASSERT (FALSE)
   REPORT "End of file detected"
   SEVERITY failure;
   WAIT;
END PROCESS;

END;
```

This is all the code needed for the testbench to fully verify the design. So all that is required to generate the testbench for any design using the package is to identify the actions that will be needed for the verification, then to give them names and specify exactly what the actions are. All that is needed now is to come up with the vectors which will test the design. With the mnemonics that were defined in the VHDL code above, the test vectors will read almost like an assembly language program. One possible test vector file that could be used to test the design above is shown below:

```
-- initialize variables
   INIT: SET term_count b(11111100)
```

```
   SET reset_cnt h(00)

-- reset and verify counter
   BEGIN: RESET
   READ
   VERIFY $reset_cnt $ERROR
   BR_SET $ERROR

-- Load counter with value near terminal count
   LOAD $term_count

-- Wait for three cycles for counter to overflow
   LOOP 3
         NOOP
         BR_SET $ERROR
   END_LOOP
   BR_SET $CONT

-- If error detected then halt simulation
   ERROR: HALT

-- Make sure that count value returns to 0
   CONT: NOOP
   BR_SET $ERROR

-- End of test
   END: ASSERT_DONE
   HALT
------------------------------------------------------------------------------------------------------
```

The package has a predefined instruction called "SET" that allows for constants to be defined and assigned an integer value. These constants can be referred to later by referencing them with a dollar sign ($) preceding the label. The labels declared at the beginning of some of the lines are also constants, and take on the integer value of the line on which they appear. They can then be used as the target of branch or jump instructions, as in the "BR_SET" instruction above.

PACKAGE COMPONENTS

The package that allows for the creation of these testbenches consists of various file, string and conversion functions and procedures, as well as type definitions for various structures which are designed to provide the VHDL model designer with a base upon which he or she can build a script-driven testbench. This collection of procedures and functions will eliminate the hassle of dealing with file I/O and string parsing routines and will allow the designer to easily create very sophisticated test code. Due to space limitations in this paper, only the procedures and functions that are significant will be described.

Types: Several types have been defined in the package to store lists, pointers and other compound structures. Many of the parameters to the functions and procedures within the package are of these defined types, so the testbench code will have to define variables of these types

before calling these procedures. Note that it is not possible to declare SIGNALs or PORTs with most of these types since the types are based on ACCESS subtypes.

String Functions: The string functions in the package can check the equality of two strings of arbitrary length, convert a string of one length to a string of another, or convert a string numeric and format characters into an integer type, Std_ULogic_Vector or Std_Logic_Vector. Each of these functions will operate on a string of any size. There is also an overloaded operator which will allow comparisons between two different strings, even if they are of different lengths.

Parse - This procedure will separate the input line, in the form of a string, into as many as four fields, the first of which is a string which represents an instruction, and the rest are integers which represent the parameters for that instruction. The first field of the string could also be a label, which is indicated by a colon in the last character position of the field. If the first string is a label, it will be ignored by the Parse procedure since the labels are taken care of when the file is read in by the Read_File procedure. Parse extracts the fields using the space or tab characters as field delimiters. If the first character of the second or subsequent fields is a dollar sign ($), then the string represents a label which should have already been defined, and the field is replaced by the value that has been assigned to that label.

Get_Integer - This procedure is called by the Parse procedure and is used to extract the next field from a string and interpret it as an integer. This could mean dereferencing a label into its contents, or interpreting the string as a decimal, hexadecimal, octal or binary value.

Get_String - This procedure is called by the Parse procedure to strip the next field from a string.

"=" - This function will compare two strings. If they are equal, it will return the value TRUE. If the strings are not equal, then FALSE will be returned. The comparison is done character by character through each string. If the characters match, then the next character is tested. If the characters do not match, then a FALSE is immediately returned. If one string is longer than the other, then the characters will be checked to the length of the shorter string. If all of the characters to that point match, then the remaining characters in the longer string are checked to see if they are all NUL characters. If so, then the strings are said to match, and a TRUE is returned. Otherwise a FALSE is returned.

File Procedures: The package also contains file procedures which automate the process of reading lines from a file and parsing them into string and integer fields. The format of the input file is described below.

Read_File - This procedure will read the entire contents of the file, creating a string for each line that is not blank, and placing a pointer for that line into the file_input variable. The number of lines that were read (not including empty lines) will be returned in the line_num parameter. This procedure will also check each line to see if it defines a label. If it does, then the Define_Variable procedure will be called to set the label's value equal to the line number of the line that contains the label. By reading the entire contents of the file into the array of string pointers, it becomes possible to create commands that loop or jump. Since VHDL does not allow random access into files, it is otherwise not possible to jump or loop within the file.

Miscellaneous Functions: The remainder of the procedures and functions in the package provide methods for defining instructions and variables which will appear in the input file. An instruction is a particular string of characters which, when read from the file, will be translated into an action or series of actions which may be applied to the pins of the device under test. Variables may also be defined in the input file which allows for the use of meaningful mnemonics in place of any of the integer fields for a command.

Define_Instruction - This procedure adds an instruction to the instruction set to make it visible to the Lookup_Instruction procedure. The number of fields required by the instruction are also stored in the set to make parsing the line containing that instruction easier.

Lookup_Instruction - This procedure will look up the given command to see if it matches any of the defined instructions in the instruction set. If a match is found, then the procedure will return the number of fields that are expected by that particular instruction. The comparison of the command against the defined instructions is done using the overloaded "=" operator which means that the command string does not need to be the same length as the defined instruction string, but could actually be longer and still match if all of the trailing characters in the string are NUL.

Define_Variable - This procedure is used to add the variable name and value pair to the list of defined variables. If the variable name already exists within the list of defined variables, then the value that is bound to that variable will be changed to the new value.

Reference_Variable - This procedure will look up the given variable name in the list of defined variables. If a match is found, then the value that is stored in the list for that particular variable will be returned. If the search is unsuccessful, then an assertion error is generated indicating a reference to an undefined variable.

CONCLUSION

The parse functions package has been in use at Philips Semiconductors since March, 1994. It has allowed designers to concern themselves more with their designs and less with the development of testbenches to verify the designs. The time that was spent on testbench development was mostly spent on the development of quality vectors and not on unproductive test fixture development. Simulation time was also reduced through the use of this package due to the nature of the vectors being self checking. Regression tests could be performed without looking at the simulation waveforms, viewing only the final results instead.

One of the limitations of the package is that it cannot be used to develop a testbench that will fully test a model unless it can be verified through its ports (i.e. it cannot sample internal nodes of a model). This is not a major limitation since most designs can be verified through their pins, but work can still be done to improve the package.