Development of a WAVES Compatible Testbench for Board-Level Test

C. Richard Unkle and William G. Swavely
IIT Research Institute
201 Mill Street
Rome, NY 13440

and

James Nagy
Rome Laboratory/ERDD
525 Brooks Road
Griffiss AFB, NY 13441

Abstract

This paper describes a generic WAVES compatible VHDL testbench that has been developed to produce test vector information that includes variable length cycles and strobe times. The test vector formats are compatible for translation to several Automatic Test Systems (ATSs) for test. The testbench is created automatically using a tool developed by the IIT Research Institute Reliability Analysis Center (IITRI/RAC) and the Rome Laboratory (RL). The tool parses a VHDL structural model and generates the testbench. The testbench uses stimulus/response data captured in the IEEE Standard 1029.1, Waveform And Vector Exchange Specification (WAVES), format. The tool was developed as part of an actual project designed to demonstrate the use of VHDL and WAVES for board-level test program set (TPS) development. A case example for an actual board model is presented. Details are also provided on the demonstration project and on specific processes that were required to make the testbench WAVES compatible. The advantages of using the WAVES format and future plans are also addressed.

1.0 Introduction/Background

IEEE Std. 1029.1-1991, The Waveform and Vector Exchange Specification (WAVES), was released as a standard for the capture and exchange of digital simulation and test vector information in 1991. Designed to be a "sister" standard to IEEE Std. 1076, VHDL Hardware Description Language (VHDL), WAVES was developed in response to a defined need in both government and industry for a standard way of representing vector information. While VHDL has continued to gain users since its release in 1987, WAVES has not. Part of this is due to the notable lack of tools for reading and writing vectors in the WAVES format. Another reason has been due to the lack of information on how to use WAVES.

In response to the need for more detailed information and examples on how to use and understand the WAVES language and standard, the US Air Force's Rome Laboratory (RL), in the fall of 1992, enlisted the support of the Reliability Analysis Center (RAC) to define and develop a tutorial on WAVES. (The RAC is a Department of Defense Information Analysis Center (IAC) operated by IIT Research Institute (IITRI) and specializing in the Assurance Technologies, such as reliability, maintainability and testability/testing.) An additional task given to the RAC was to support an effort at the San Antonio Air Logistics Center (SA-ALC) to demonstrate the use of VHDL and WAVES in the development of digital, board-level, Test Program Sets (TPS). Additional efforts are currently being sponsored by RL in the area of WAVES user guides and development tools.
1.1 The TPS Demonstration Program

The effort supported by RL and IITRI/KAC was to assist SA-ALC in re-hosting two existing TPSs, one for each of two printed circuit boards (PCBs), from an older Automatic Test System (ATS) to a GenRad 275X series ATS, and an ATLAS based government tester. Because the goal was to demonstrate the use of both VHDL and WAVES, the TPSs were developed by re-describing the existing board designs in VHDL, and then performing simulation to develop new test vectors. Stimulus vectors, taken from the existing TPS test vectors, were to be captured in WAVES, and then while in the WAVES format, used to drive the simulation. The remainder of this paper will concentrate on describing the details of the design of the testbench used for test vector development, and a prototype software tool that was created to automatically produce the testbench, based on a VHDL structural file of the PCB, and defined WAVES standard names for logic values. For more detailed information on the entire VHDL TPS demonstration program, see references [1]-[3].

The WAVES Port List is the mechanism built into the current set of WAVES standard packages that enables information from WAVES to be passed to other environments, such as a VHDL testbench. Understanding the WAVES Port List data structures is key to development of a working WAVES compatible VHDL testbench. Figure 2.1 illustrates elements of the WAVES Port List which are important to this discussion.

This field is an array from 1 to n where n is the number of primary input/output (I/O) signals in the Device-Under-Test (DUT) and defined in a WAVES 'test pin declaration' file. The order in which the test pins, or primary I/O signals are declared in the WAVES test pin declaration file must match the order the signals appear in the WAVES External file. In accordance with the WAVES standard, stimulus and/or response vector data (e.g., 1's, 0's, L's and H's) can reside in an external file. Timing data, in the form of waveform slice information, can also be included in the external file, as they were for the TPS development program. Each array element in the .WPL field corresponds to a column in the External File.

2.0 Understanding The WAVES Port List

![Diagram of WAVES Port List]

**FIGURE 2.1: ILLUSTRATION OF THE WAVES PORT LIST**

6.2
The array element ‘L_Value’ in Figure 2.1, is a set of integer values from 0 to n, where n is the total number of logic value names declared in the appropriate WAVES package. The integer order, (i.e., 0, 1, 2, ...) corresponds to the order in which the logic value names are declared, as depicted in Figure 2.1. When the Logic Values are declared in the WAVES Logic Package Declaration, the position of each Logic Value is assigned an integer starting from 0 and ending with the number of Logic Values minus 1. It is this integer value information that gets passed through the WAVES port list to the VHDL testbench environment. This is accomplished via execution of the 'Waveform Generator Procedure' (WGP). The WGP is a procedure defined within the WAVES dataset that will read the external file, decode the pin codes into logic values, and pass this information onto the testbench through the WAVES Port List. In this manner, pin codes are applied to a signal in a print on change format. The timing information is also passed to the testbench so that simulation can proceed accordingly.

As an example, suppose array element 1 of the .WPL corresponds to the clock pulse signal in a DUT, and suppose that two Logic Values are declared named “DRIVE_1” for a logic level high and “DRIVE_0” for logic level low. Finally, suppose that these Logic Values were declared in the first and second positions of the Logic Value Declaration string:

```
package DUT_WAVES_LOGIC is
type Logic_Value is
  (DRIVE_1, DRIVE_0, SENSE_1, SENSE_0, ...)
```

These Logic Values would receive integer place holders of “0” for DRIVE_1, “1” for DRIVE_0, “2” for SENSE_1 and so forth. When a line of pin codes, called a ‘slice’ (of the total waveform) is read within the Waveform Generator Procedure, each pin code is broken down, a logic value at a time, converted to the corresponding integer value, which is then passed to the testbench. Within the testbench, procedures were written to translate the Logic Values into IEEE standard 1164 Logic Values, prior to being applied to the appropriate signal.

2.2 Waveform Generator Procedure (WGP)

The WGP essentially uses all of the definitions of logic values, pin codes, etc., defined within the WAVES dataset to interpret and apply the vector information contained in the external file. The data within the external file is read in a line, or slice, at a time. There is a total of 901/O pins for the Relay Driver Card; the ‘L’s and ‘H’ pin codes represent expected response information on the board’s output pins.

As a line in the external file is read in, the information is applied using the WAVES ‘apply’ procedure to the signal CONNECT, which is of type WAVES_PORT_LIST. The information applied to the WAVES_PORT_LIST is the pin codes and timing data.

3.0 TESTBENCH DESIGN

As part of the TPS development for digital systems, test vector information must be developed via simulation. Vectors must be developed that provide a performance test of the DUT. Typically, test vectors are developed that will provide a predefined level of fault detection. This is accomplished by first developing vectors via good circuit simulation, and then ‘grading’ these vectors for their fault detection capability via fault simulation. In this case, fault simulation was not possible, since this capability is not yet available in VHDL. However, for the demonstration project, an existing set of known good test vectors were already available, which were captured in WAVES, and used to both verify the VHDL model of the boards to be tested, and to generate both performance and diagnostic test vectors. For digital boards, there are two possible methods of fault diagnostics; probing and fault dictionary. Probing is accomplished by capturing internal board node data during simulation, and using this as a response database within the Automatic Test System (ATS). Many ATS have algorithms that will determine the order in which probing is to be performed, based on a description of the board (usually a netlist). The actual data at a particular node is compared to the stored information developed via simulation. The other method of diagnostics is to generate a fault dictionary, which is an ordered set of information consisting of fault signatures, and
the possible faulty components that could have resulted in those signatures. Even though this was not an option for this demonstration, fault dictionaries are not always used as they are costly to develop, and require large amounts of memory to store on the ATS.

Given that fault simulation was not an option, the testbench was designed to capture I/O information, based on known good stimulus vectors for each board. The I/O information was used for the performance test. The testbench was also designed to capture all internal, as well as external, node information to be used as a probing database for diagnostics. In addition to the vector information, timing data is also captured as an output of the testbench. The output of simulation is a print on change format. However, for test, vector information needs to be ‘cylized’, wherein a set of signals all have the same timing. For many older testers, this meant that all vectors were applied within a constants cycle time, such as 1000ns cycles. For many of today’s testers, variable timing cycles can be defined. The need to take vectors that are not in cycles and convert them to vectors that are cycliced is one of the major challenges to the world of digital test. However, in this case, the vectors already existed in a cycled format. Because of this, the testbench was also developed to capture the existing cycle time information. This was of some help, however many testers have their own rules regarding timing and cycle definition.

For the demonstration project, both cards had existing test vectors applied on a MATE tester. These vectors were in equal cycles of 1000ns. One of the testers this information was to be ported to is a GenRad 275X tester that had different ‘rules’ for vector cycle definition. Therefore, a translation program was written to take the vectors produced via the testbench to be discussed, and put them in a format acceptable to the GenRad. Further details of this can be found in reference [1].

3.1 Testbench Routines To Interpret WAVES Data

As discussed, routines were written to interpret the incoming WAVES data supplied via the WAVES apply procedure onto the WAVES_PORT_LIST. As previously described, the WAVES data are integer values that represent user defined logic values. Procedures were developed that translate these logic values into IEEE Standard 1164 logic values that are then applied to the appropriate signal within the VHDL model. The line below is an example of how the WAVES information was applied to a signal in the testbench.

<signal_name> <=To_1164(Logic_value'val(
  WAVES_DATA.WPL(1)L.VALUE));

The signal ‘WAVES_DATA’ is declared as type WAVES_PORT_LIST and used to import information applied to the WAVES_PORT_LIST by the Waveform Generator Procedure described earlier. The function within the innermost parentheses determines the integer value contained in the ‘L.VALUE’ field of the WAVES_PORT_LIST (.WPL(1) in the example) for the signal <signal_name>. The function Logic_value’val, then converts the integer back into the logic value name defined by the user. For instance, in Figure 2-1, the integer value of ‘0’ corresponds to the logic value name ‘DONT_CARE’, ‘1’ to ‘DRIVE_0’, ‘2’ to ‘DRIVE_1’, etc. The To_1164 function then converts the logic value names to IEEE standard 1164 logic values. The To_1164 function is provided as Figure 3-1.
The Logic_Value names listed in the above procedure were standard names chosen for the demonstration project. This allowed the above code to remain the same for any board model to be simulated, eliminating the need to develop a tool that could read a WAVES dataset, and determine logic value names and their meanings automatically.

**3.1.1 BI-DIRECTIONAL SIGNALS**

The use of WAVES logic value names enabled bi-directional signals to be handled very easily within the testbench. In many instances, bi-directional signals are divided into two separate signals within the testbench, one for inputs and one for outputs. Since the logic value names created by the user have meanings associated with them, including direction, the same signal can be used for both input and output. This was handled by creating a procedure called 'TO_DRIVE'. The TO_DRIVE function is called as part of the WAVES translation process defined previously. As an example:

```lisp
P1X16<=TO_DRIVE(Logic_value'val(WAVES_DATA.WPL(1).L_VALUE));
```

where the TO_DRIVE function is defined:

```lisp
function TO_DRIVE( VALUE2 : Logic_value ) return Std_logic is
begin
  case VALUE2 is
    when UNKNOWN => return 'Z';
    when UN_INT  => return 'Z';
    when HI_IMP  => return 'Z';
    when DONT_CARE => return 'Z';
    when DRIVE_0  => return '0';
    when DRIVE_1  => return '1';
    when SENSE_0  => return '0';
    when SENSE_1  => return '1';
  end case;
end TO_DRIVE;
```

Basically, all stimulus signals are defined with pin codes comprising Drive_0 or Drive_1 logic values, and all response signals are defined with pin codes comprising Sense_0 and Sense_1 logic value names. This puts a burden on the designer to know when the bi-directional signal is driving versus when it is expecting response. When the signal is driving, the appropriate IEEE standard logic level is applied. Whenever the signal is being driven by the simulation, the incoming WAVES information is translated to a logical 'Z', which creates a conflict on that signal that will always be resolved to the simulation value via the IEEE Standard Logic 1164 resolution function (i.e. a 'Z' and 'any other logic value', always resolves to 'any other logic value')
3.1.2 Nodal Capture Routines

As mentioned before, the objective of the testbench was to capture all signals at each node, both internal as well as external, to be used as performance test vectors and a probe database for diagnostics. Therefore, processes were created that capture this information, including timing data. An example of the capture process for a signal named 'clk' is provided in Figure 3-2. The capture process writes the state of the signal, and the time at which the state occurs. The state is determined by the 'TO_CHAR' function shown:

function TO_CHAR(
VALUE:in std_logic;DIRECT:character
return character is
begin
if DIRECT = '1' then
  case VALUE is
  when '0' => return '0'
  when '1' => return '1'
  when 'X' => return 'X'
  when 'Z' => return 'U'
  when 'U' => return 'U'
  when 'X' => return 'U'
  when 'L' => return 'L'
  when 'H' => return 'H'
  when '0' => return '0'
  when '1' => return '1'
  when 'X' => return 'X'
  when 'Z' => return 'U'
  when 'U' => return 'U'
  when 'L' => return 'L'
  end case;
else

end if;
end TO_CHAR;

The capture routines have been structured to write '1's and '0's when the signal is an input, and 'H's and 'L's when it is an output. Further, any 'Z's are converted to 'U's. The reasons for Ls and Hs, and converting Zs to Us has to do with the world of ATS. ATSs typically distinguish response values in this manner, and many cannot typically sense a 'Z' value.

While there were other nuances of the testbench that were required, they are not relevant to this discussion. Additional details can be found in references [1] and [2].
4.0 Creating The Testbench Automatically

Because several designs were to be simulated as part of the demonstration project, no one had the desire to manually create multiple testbenches. Therefore, once the testbench format was finalized, a program was written to automatically develop the testbench based on a number of inputs. For the purposes of the demonstration project, specific rules and naming conventions were defined, such that a testbench writer tool could be quickly and easily prototyped. The process of creating the testbench and capturing the desired output is described by Figure 4-1. Details on the required inputs and naming conventions are provided below.

**Required Input Files**

The files necessary to produce the testbench are represented by the boxes with an 'i' in the lower right corner. These files are:

- A working VHDL model and corresponding WAVES dataset. The VHDL model must be structural, and have been developed and exercised in a commercial-quality VHDL environment prior to input to the testbench writer tool. This eliminated the need to have extensive language specific error checking in a prototype tool.

- A power supply information file with the '.POW' extension. This file contains information on the power and ground pins on the board to be simulated. While this information is typically not needed for simulation, it is needed for test purposes, and is therefore included in the testbench.

In addition to the above input files, the following rules and conventions were established to enable rapid prototyping:

- All signals used within the VHDL model must be of type STD_LOGIC or STD_LOGIC_VECTOR.

- The external I/O signals must be placed in the port declarations section of the testbench entity, and therefore the VHDL model, in the same order as they appear in the WAVES external vector file. This ensures that the WAVES Port List is correctly connected to the testbench signals.

- The VHDL model must contain structural references only. However, the component models called the VHDL model may be behavioral. All model initialization routines must be compiled in the component models, and not in the input VHDL model.
CAP_PIX16_STROBE:
process(W_PIX16)
variable VALUE3 : Logic_value;
variable DIRECT : character;
variable LOUT : line;
variable SIG_NAME : string(1 to 6);
variable STATE : character;
variable SIM_TIME : time;
variable SPACE : character := ' '; 
begin
VALUE3 := Logic_value'val(WAVES_DATA.WPL(1).L_VALUE);
DIRECT := FIND_DIRECT(VALUE3);
SIG_NAME := "P1X16";
STATE := TO_CHAR(W_PIX16,DIRECT, VALUE3);
if STATE = 'H' or STATE = 'L' or STATE = 'X' then
  STATE := '1';
else
  STATE := '0';
end if;
SIM_TIME := now;
write(OUT, SIG_NAME);
write(OUT, SPACE);
write(OUT, STATE);
write(OUT, SPACE);
write(OUT, SIM_TIME/\ns);
writeln(CAPTURE_FILE, LOUT);
end process;

FIGURE 4-2: STROBE SIGNAL CAPTURE ROUTINE

- The WAVES dataset must contain the logic value names used by the TO_1164 function within the testbench (these routines are currently 'hard-coded' into the testbench.)

Additional rules concerning extra signal or pin names for capturing of any existing vector cycle information and test strobe data were also established. The strobe data is necessary to tell the ATS when an output signal from a device is to be sensed. At any other time, the signal is a 'don't care'. The demonstration project used the response pins within the WAVES external file to capture strobe data. In this manner the testbench capture file automatically associates a strobe signal with every output or bi-directional pin (as determined from the signal declarations in the VHDL model). The signal name is written to the testbench output file, followed by the same signal name on the next line, however preceded by an asterisk (*). For example, signal OUT_A is the output pin, and *OUT_A is the associated strobe. The strobe signals in the output file will pulse high at the desired strobe time (determined by the design and/or test engineer), and remain a logic low otherwise. By matching the signal name with its *name, enable the post-processing tools created for the project to recognize when an output was to be strobed, and pass this information on to the ATS. An example capture process, showing the strobe signal is provided in Figure 4-2. In reviewing Figure 4-2, the strobe signal (P1X16_STROBE), will process on a signal called W_PIX16, which is a signal created to read the incoming WAVES data to see if a strobe is required for signal P1X16. This is possible by converting the .WPL information to a logic value name, as is done in the TO_1164 and TO_DRIVE functions, and acting on this information in the function called FIND_DIRECT and TO_CHAR. These functions are shown in Figure 4-3 and 4-4, respectively. Note that the FIND_DIRECT function is only necessary for Bi-directional signals. For output only signals, FIND_DIRECT is not used. The TO_CHAR function is actually
an overloaded function as this was defined previously for input signals in Section 3.1.2.

```plaintext
-- Determine direction of signal for captured output (BIDIRECTIONAL I/O)
--
function FIND_DIRECT( value3 : Logic_value ) return character is
begin
  case value3 is
    when UNKNOWN  => return 'O';
    when UN_INT   => return 'O';
    when HI_IMP   => return 'O';
    when DONT_CARE => return 'O';
    when DRIVE_0  => return 'T';
    when DRIVE_1  => return 'T';
    when SENSE_0  => return 'O';
    when SENSE_1  => return 'O';
  end case;
end FIND_DIRECT;

FIGURE 4-3: FIND_DIRECT FUNCTION FOR BI-DIRECTIONAL SIGNALS

-- Overloaded convert STD_LOGIC to a character
function TO_CHAR(VALUE:std_logic;DIRECT:character; DRIVE_VALUE:logic_value) return character is
begin
  if VALUE = 'X' then
    case DRIVE_VALUE is
      when DRIVE_0  => return 'C';
      when DRIVE_1  => return 'D';
      when others    => return 'X';
    end case;
  else
    if DIRECT = 'I' then
      case VALUE is
        when '0'  => return '0';
        when '1'  => return '1';
        when 'Z'  => return 'U';
        when 'U'  => return 'U';
        when '-'  => return '-';
      end case;
    else
      case VALUE is
        when 'L'  => return 'L';
        when 'H'  => return 'H';
        when '0'  => return 'L';
        when '1'  => return 'H';
        when 'Z'  => return 'U';
        when 'U'  => return 'U';
        when '-'  => return '-';
      end case;
    end if;
  end if;
end TO_CHAR;

FIGURE 4-4: OVERLOADED TO CHAR FUNCTION

6.9
5.0 SUMMARY AND CONCLUSIONS

The main purpose of this paper was to provide the reader with information important to understanding how WAVES can be integrated with VHDL in a testbench environment.

The demonstration of using WAVES with VHDL was performed on an actual project wherein existing TPSs for digital PCBs were re-hosted to different ATSs. This process included recapturing the board designs in VHDL, and using the existing test vectors to validate the VHDL designs and to re-generate test vectors that were ported to the target ATS. These existing test vectors were captured and documented using a WAVES Level I dataset. Further, naming conventions and VHDL modeling styles were defined that enabled tools to be written that can automatically create a VHDL WAVES compatible testbench designed to generate information needed to functional test each board and provide a probing database for diagnostics.

The primary purpose of the described demonstration project was to show that board level TPSs could be developed using standard formats, including VHDL and WAVES. This demonstration was successful. However, further research is needed in the area of how using WAVES affects overall simulation efficiencies, and how WAVES routines need to be structured to improve simulation efficiency. Further, for VHDL to be a viable alternative to development of digital test, VHDL fault simulation tools must be developed. Rome Laboratory is currently in the process of a program to specify an environment and set of software tool capabilities that can provide for fault injection, concurrent fault simulation of models containing both behavioral and structural components, test generation, fault grading, and other test applications based on VHDL.

6.0 REFERENCES


