

Testview: An Innovative Approach to Automatic VHDL Testbench Generation

Steve Smith
Microsystems Prototyping Laboratory
Engineering Research Center
Mississippi State University
Phone: (601) 325-7560

Abstract

The VHISIC Hardware Description Language (VHDL) is a formal notation intended for use in all phases of design of electronic systems and has become an accepted standard [1]. While much progress has been made in formulating the language, and making this standard more adaptable to the needs of the microsystem design community, the language does not directly address issues relating to VHDL model verification and test. Unfortunately, the manual generation of a stand-alone, design specific VHDL test environment that would enable the model verification process as a function of the model development cycle can be as labor intensive and time consuming as the development of the model itself. Consequently, the task of testbench generation is often viewed as a peripheral task, something to be tacked onto (rather than integrated into) the model development process.

This paper presents a highly configurable automatic VHDL testbench generation methodology (Testview) in support of VHDL model verification and test at any level and phase of the modeling process. Innovative features of this integrated VHDL testbench methodology include:

- support of abstract data types at the I/O boundary
- library management of testbench architectural components and user interface modules
- dynamic generation of user interface for support of testbench options and parameters

Introduction

The primary purpose of the Microsystems Prototyping Laboratory of Mississippi State University is the establishment of a VHDL center of excellence to address current and future government and industry research and development needs in the area of VHDL modeling, model verification, and microsystem design and redesign activities.

The automatic VHDL testbench generation methodology currently underway within the MPL consist of three principle components. They are the testbench architecture development strategy, the implementation development strategy, and the user interface development approach. The architecture development approach consist of the coarse grain partitioning of the VHDL testbench into architectural component modules that can be instantiated into the testbench proper based on user needs and design requirements. Component modules are grouped according to functionality within component classes. Component classes are created and maintained via the Testview library management utilities. New components may be added to the system and linked to the library manager for subsequent use. The component modules are implemented as parameterized component generator templates. The generator utilities operate on the component generator templates to generate MUT (model under test) specific testbench components using information supplied by the user via the front-end user interface. The back-end user interface is generated

concurrently with the MUT specific testbench to allow the user easy access to the test options associated with each of the component modules that have been instantiated within the testbench.

A nine word by twelve bit stack component module used in the model for an off the shelf commercially available microprogram controller will be used to illustrate the VHDL testbench generation strategy.

The Stack I/O Interface

As electronics system complexity increases, more and more sophisticated means are being required for capturing and modeling their behavior. VHDL offers data abstraction capability beyond that of simple logic states and allows models and test programs to be generated that can help shorten the design cycle of electronic modules[2]. Data abstraction offers the advantage of translating the normal binary interface level to the model into a more human readable form. The example stack component illustrates the data abstraction concept at the interconnect level of the model. The I/O interface for the stack module is listed in Figure 1.

```
port(  
    Dout : out std_ulogic_vector(11 downto 0), — data out  
    Din : in std_ulogic_vector(11 downto 0), — data in  
    Stkptr : in std_ulogic_vector(3 downto 0), — stack pointer  
    Clk: in std_ulogic, — clock  
    Stkcmd: in stk_type, — stack command PUSH, POP, HOLD, CLEAR  
);
```

Figure 1 : Stack I/O Interface

A VHDL enumerated data type was created for the Stkcmd port for the stack component. This data structure adds clarity to the simulation environment by allowing the user to easily interpret the command as described by the data sheet for the part. In order to successfully generate the testbench for this example, the Testview utilities must synthesize the necessary interface and conversion information for all Testview component modules for which this type is visible. The allowed values of the enumerated type `stk_type` used in the example is commented in Figure 1.

Testview Architecture Development Strategy

The methodology developed to support VHDL model testbench generation is based on an architectural synthesis approach which is driven by user and design requirements. The modular decomposition of testbench components combined with an architectural synthesis approach provides for maximum flexibility and configurability for defining design specific testbench requirements. As illustrated in Figure 2, a key component of the Testview philosophy is the organization of testbench architectural component modules into component class libraries.

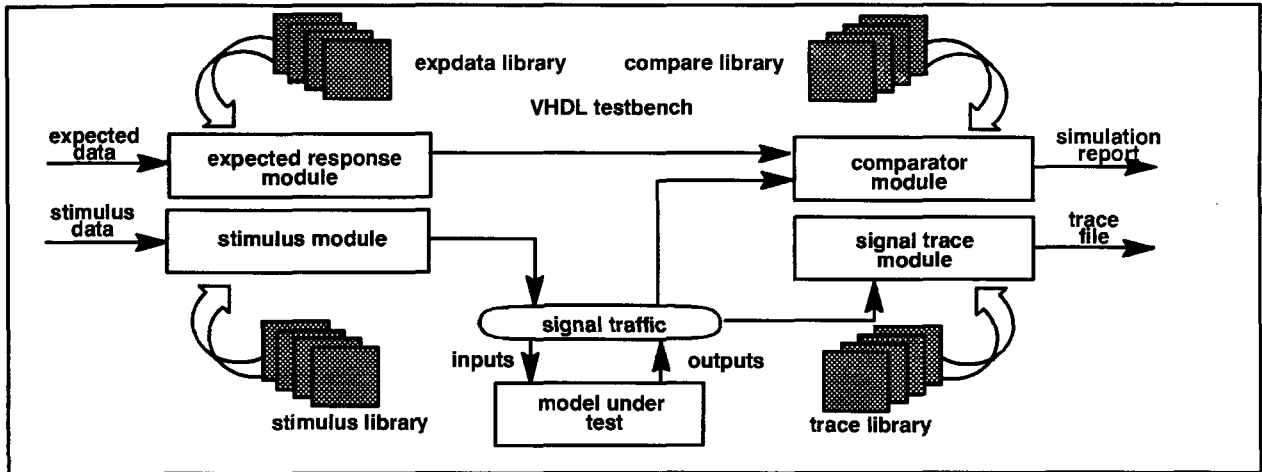


Figure 2 : Testview Architecture Development Approach

The class objects and testbench structure illustrated in Figure 2. depicts a basic test environment for applying stimulus and comparing actual simulation results with expected results for the model under test. The Testview library and component manager utilities provide the necessary mechanisms for defining and maintaining class libraries. Each component class library contains the necessary information for interfacing with the model under test as well as other component classes. Testview provides the user with a menu driven front-end for selecting the appropriate components for testbench generation. New component modules may be added to the library class structure as needed. If for example, model test requirements cannot be met using existing library components, component modules can be built and added to the appropriate library via the library management utilities for subsequent use. Once testbench requirements have been specified for a particular model, the Testview synthesis utilities will select the appropriate library components and generate a stand-alone ASCII readable VHDL testbench that may be utilized throughout the entire model development cycle.

Testview Implementation Development Approach

The block diagram for the Testview implementation strategy is shown in Figure 3. The Testview testbench user interface is optional from the users' perspective as indicated by the dashed lines. This interface allows the user some degree of test control by setting testbench component options and parameters.

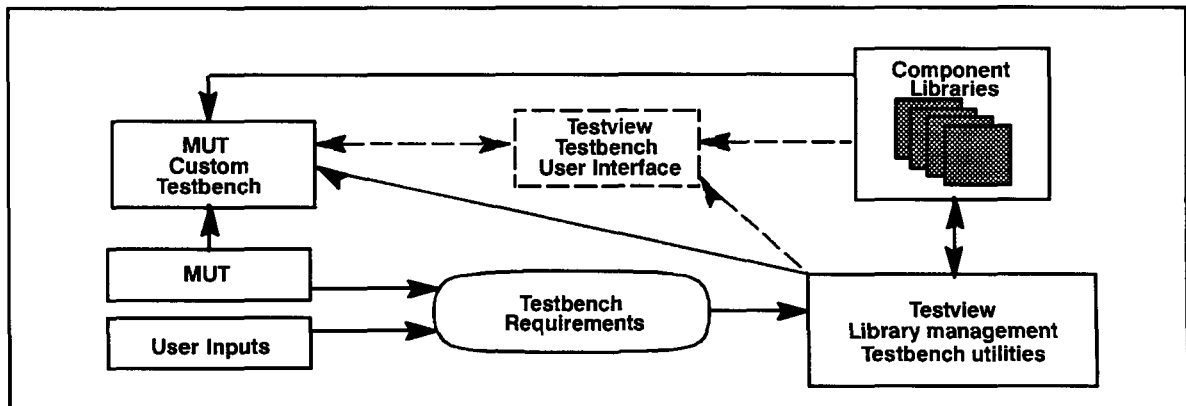


Figure 3 : Testview Implementation Approach

In order to present the implementation of Testview component modules, this section will address the implementation of a stimulus component used to test the stack component given as the example. The stimulus vector file contains test vectors without any corresponding timestamp information. The timing information is added to the stimulus stream internal to the stimulus module relative to an internal strobe signal. The user sets the leading and trailing edges of the stimulus stream prior to simulation via deferred constants in a package body. Figure 4 shows an example input stimulus and corresponding waveform for the Clk input of the stack module with the timing and mode options selected.

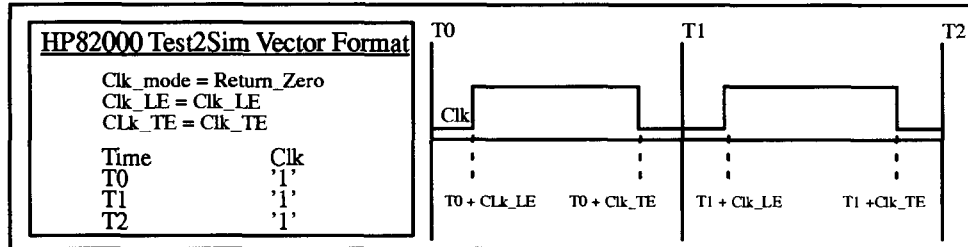


Figure 4 : Test2Sim Component Module Vector Format

The VHDL component architecture for the stimulus module is shown in Figure 5. As was mentioned previously, all component modules are represented as generator templates. In this example, the foreach statement is interpreted by the generator utilities as an iterative construct and builds an MUT specific stimulus architecture using the parameters highlighted in bold type.

```

architecture test2sim of stimulus_port is
  signal mclk : std_ulogic := '1';
  foreach i in PINLIST generate
    signal PINLIST_s : PORTTYPE;
  endfor
begin
  mclk <= not mclk after mcycle;

  getstimulus(
    foreach i in PINLIST generate
      PINLIST_s,
    endfor
    mclk,
    header_block);
  foreach i in PINLIST generate
    merge(PINLIST_le, PINLIST_te, PINLIST_mode_select, mclk, PINLIST_s, PINLIST);
  endfor
end test2sim;

```

Figure 5 : Test2Sim VHDL Testview Component Architecture

Using this approach, the generator utilities may be endowed with any number of interpretive features to build MUT specific testbench components of arbitrary structure and complexity.

The entity declaration for the stimulus component module is generated by the Testview netlisting utility and is parsed by the generator utility to satisfy the parameters **PINLIST** and **PORTTYPE** of Figure 5. The component architecture generated for the stack module is shown in Figure 6.

```
architecture test2sim of stimulus_port is
constant mcycle : time := 0 ;
signal mclk : bit := '0' ;

signal Din_s : std_ulogic_vector(11 downto 0) ;
signal Stkptr_s : std_ulogic_vector(3 downto 0) ;
signal Clk_s : std_ulogic ;
signal Stkcmd_s : stk_type ;

begin
    mclk <= not mclk after mcycle ;
    getstimulus(Din_s, Stkptr_s, Clk_s, Stkcmd_s, mclk, header_block) ;

    merge(Din_le, Din_te, Din_mode_select, mclk, Din_s, Din) ;
    merge(Stkptr_le, Stkptr_te, Stkptr_mode_select, mclk, Stkptr_s, Stkptr) ;
    merge(Clk_le, Clk_te, Clk_mode_select, mclk, Clk_s, Clk) ;
    merge(Stkcmd_le, Stkcmd_te, Stk_mode_select, mclk, Stkcmd_s, Stkcmd) ;
end test2sim ;
```

Figure 6 : Example Stack Stimulus Component Architecture

While the stimulus component architecture was used to illustrate the Testview implementation strategy, a number of other files are also required to reside in the Testview component library. These include header files, VHDL package definitions to support component parameters, and VHDL package definitions to support MUT specific functions and procedures. An optional file which resides in the component library supports the Testview parameter user interface and is discussed at some length in the following section.

Testview User Interface

It was decided early on in the testview development cycle that a flexible and configurable user interface would be needed in order to efficiently manage testbench options and parameters while at the same time maintain the flexibility of the Testview environment. For example, in the stimulus component architecture of Figure 6., there are three user selectable options for each MUT input. As the number of MUT inputs increase, the number of user options can become quite high.

The mechanism chosen for defining the user interface for a MUT specific testbench is virtually identical to that which defines the testbench architecture for the MUT. That is, the definition of a generator template (Testview view specification) which resides within the same directory as the component module it represents. The Testview view specification for the MUT is defined by a context free grammar. The user interface for the model is generated concurrently along with the testbench architecture by parsing the Testview view specification for the model.

The Testview generator template which complements the stimulus component architecture of Figure 5 is shown below in Figure 7.

```

VIEW input_timing_params {
  PARAMETERS initialize All_pins (
    STRINGS Stimulus_file,
    STRINGS Expect_data,
    INTEGER max_cycles,
    TIME mcycle NS );
  for each i in PINLIST generate
  PARAMETERS input_timing PINLIST (
    ENUMERATION PINLIST_mode_select (R1,R0,RZ,RD,RX),
    TIME PINLIST_le NS,
    TIME PINLIST_te NS );
  endfor
}

```

Figure 7: Testview View Specification Generator Template

The key-words for the Testview view specification are highlighted in bold type. This method of defining the Testview user interface offers some distinct advantages over other "hard wired" approaches. By describing the UI in ASCII form for new component modules added to the Testview libraries, the need for developing additional windowing applications and utilities to support the new components is eliminated. Figure 8 shows the user interface for the example stack module in an active screen configuration.

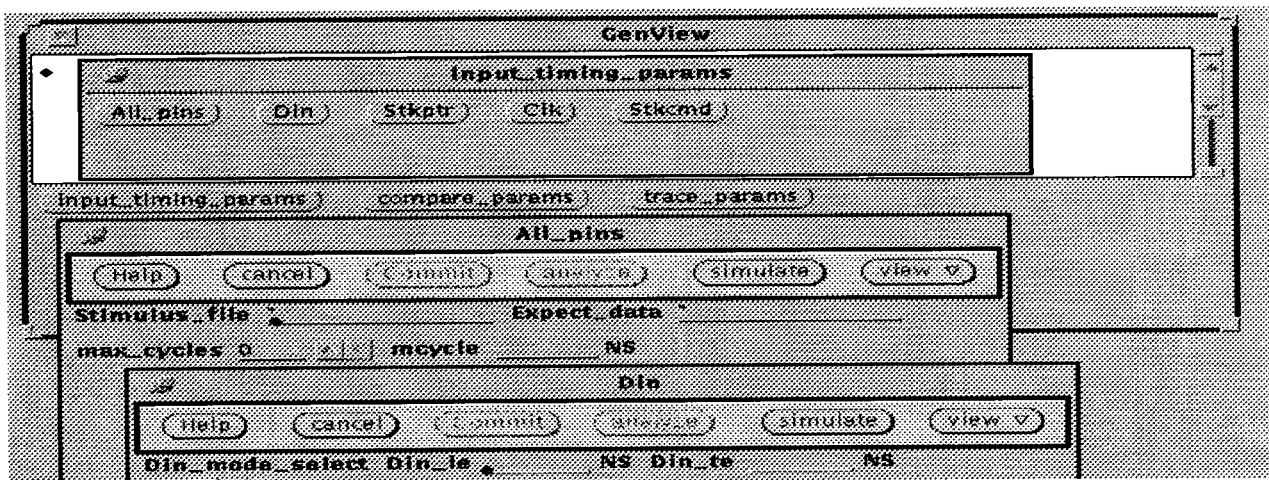


Figure 8: Testview UI For Example Stack Module

The Testview user interface software and utilities were developed under the OPEN LOOK User Interface. The OPEN LOOK UI was chosen for this project as it offers some significant advantages over other user interface environments. These advantages include:

- good visual design
- device/platform independence
- interoperability with other popular user interfaces

Figure 9 shows the partial listing for the Testview view specification used to generate the UI for the example stack module.

```
VIEW input_timing_params {
  PARAMETERS initialization All_pins (
    STRINGS Stimulus_file,
    STRINGS Expect_data,
    INTEGER max_cycles,
    TIME mcycle NS );
  PARAMETERS input_timing Din (
    ENUMERATION Din_mode_select (R1,R0,RZ,RD,RX),
    TIME Din_le NS,
    TIME Din_te NS );
  PARAMETERS input_timing Stkptr (
    ENUMERATION Stkptr_mode_select (R1,R0,RZ,RD,RX),
    TIME Stkptr_le NS,
    TIME Stkptr_te NS );
}
VIEW compare_params {
  PARAMETERS compare_params Dout(
    ENUMERATION Dout_Mode (INHIBIT, STROBE, WINDOW, PREDICT, RESPONSE),
    TIME Dout_win_le NS,
    TIME Dout_win_width NS,
    TIME Dout_strobe_point NS );
}
VIEW trace_params {
  PARAMETERS trace_params TRACE_PARAMS (
    STRINGS Trace_File_Name ,
    ENUMERATION Trace_Mode(INHIBIT, STROBE, POC),
    TIME trace_strobe NS );
}
```

Figure 9: Stack Module Testview View Specification (Partial Listing)

Conclusions

The automatic testbench generation methodology under development in this project addresses a number of critical VHDL modeling issues. Testbenches generated using this methodology do not depend on any particular data formats or simulators ensuring test portability across multiple simulation platforms. The testbench methodology described in this paper may be used to facilitate the model verification process as well as to generate VHDL model testbenches suitable for delivery to the end customer.

Future Work

The Waveform and Vector Exchange Specification (WAVES) is a direct application of VHDL which was developed out of an effort to define and standardize VHDL usage for test applications. The primary purpose of WAVES is to define a common data format which can be used for the exchange of stimulus and response and test information among simulators and testers[3]. As of this writing there are few tools

available which can facilitate the integration of WAVES into a structured test environment.

Future work involving the Testview methodology development project include the incorporation of the WAVES IEEE 1029.1 standard to facilitate VHDL model test for models using any arbitrary signal type. The MPL is also investigating the merits of a WAVES compatible intermediate form for stimulus/response datasets along with the necessary translation utilities to translate datasets from specific test and simulator platforms.

Acknowledgments

This project is funded by DARPA. The author wishes to thank all the researchers who have made valuable contributions to this work.

References

- [1] "IEEE Standard VHDL Language Reference Manual (VHDL-LRM) IEEE Std 1076-1987", 1988, The Institute of Electrical and Electronics Engineers, Inc. New York, NY
- [2] J Scott Calhoun, "Abstract VHDL Data Types As Applied To Specification Modeling", VHDLInternational Users' Forum Conference, Spring 1990
- [3] IEEE Standard for Waveform and Vector Exchange (WAVES) IEEE Std 1029.1-1991", 1991, The Institute of Electrical and Electronics Engineers, Inc. New York, NY