



VHDL Testbenches And Basic WAVES Topics

RASSP Education & Facilitation Module 61

Version 3.00

Copyright 1998 University of Virginia
This module was created under Air Force Contract #95-C-0220.

Copyright © 1995-1999 SCRA

All rights reserved. This information is copyrighted by the SCRA, through its Advanced Technology Institute, and may only be used for non-commercial educational purposes. Any other use of this information without the express written permission of the ATI is prohibited. Certain parts of this work belong to other copyright holders and are used with their permission. All information contained herein may be duplicated for non-commercial educational use provided this copyright notice is included. No warranty of any kind is provided or implied, nor is any liability accepted regardless of use.

The United States Government holds "Unlimited Rights" in all data contained herein under Contract F33615-94-C-1457. Such data may be liberally reproduced and disseminated by the Government, in whole or in part, without restriction except as follows: Certain parts of this work belong to other copyright holders and are used with their permission; This information contained herein may be duplicated only for non-commercial educational use. Any vehicle, in which part or all of this data is incorporated into, shall carry this legend.

Copyright © 1995-1999 SCRA

1



Module Goals



- **Introduce VHDL Testbenches**
 - Used To Verify And Test VHDL Designs
- **Introduce WAVES**
 - Basic Concepts
 - WAVES Library And Functions
 - WAVES Test Set
 - Basic WAVES Examples

Copyright © 1995-1999 SCRA

2

The goals of this module are to first introduce VHDL testbenches and explain their use during the testing process. Then, WAVES is described through basic concepts, waveform functions, and examples. It is expected that the student gain an appreciation for WAVES without having to review many of the implementation details that support WAVES. After completing this module, the student should be able to create a VHDL testbench, write a WAVES waveform generator file, write a WAVES external test vector file, and complete a successful simulation of a WAVES test set. It is assumed that the student is familiar with the details of the VHDL language. Since WAVES is a subset of VHDL, the student should be able to follow the syntax and examples presented in this module.



Module Outline



● Introduction

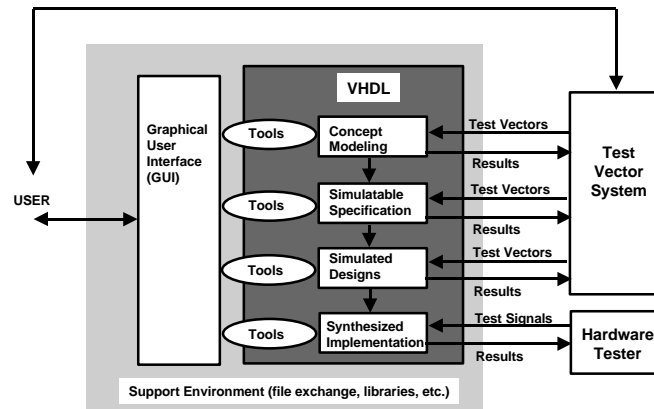
- Testbench Development
- Design Verification Challenges
- WAVES Concepts
- WAVES Constructor Library and Built-Ins
- WAVES External File
- WAVES Test Set
- Decoder Example
- Algorithmic Waveform Generator Example

Copyright © 1995-1999 SCRA

3

Introduction

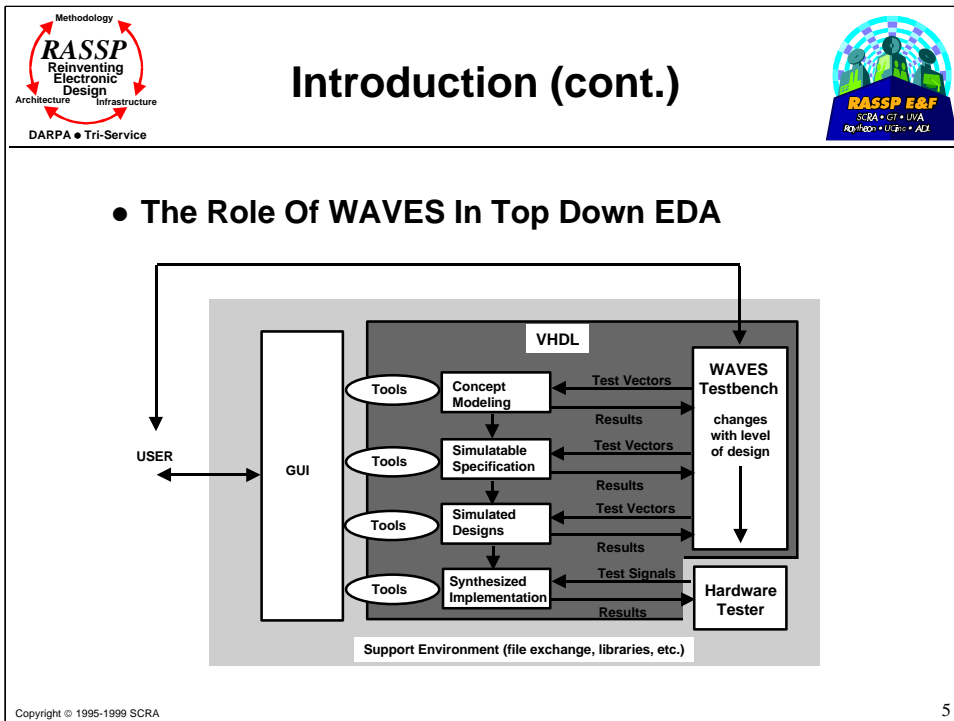
• Top Down Electronic Design Automation (EDA)



Copyright © 1995-1999 SCRA

4

Most EDA tool developers are using VHDL as the underlying engine beneath their tool suite. Using VHDL, a design can be simulated at any level, from concept to implementation. However, the unification of the EDA tools around VHDL has created a requirement. The designer needs to be able to stimulate the simulations at the various stages of development and to collect the results of these simulations. In other words, the designer needs test vector generation and results collection and comparison for the simulated development descriptions at all stages. WAVES was created to meet this need. [Hanna97]



WAVES was designed to be the unified testing and results collection system to complement the unified development systems based on VHDL. Its purpose is to provide the means to define test stimuli, in the form of digital waveforms (or test vectors), to define the results to be collected, and to manage the insertion of the stimuli and the collection and comparison of the results as the VHDL description is simulated. It is also designed for compatibility with hardware testers, such that the same test stimuli and collection paradigm may be automatically communicated to hardware test systems. This ensures identical testing of the hardware and pre-implementation simulation. The testing and collection entity of WAVES, called the WAVES testbench, is written in VHDL and attached to the VHDL description. The testbench is analyzed, compiled and executed along with the rest of the VHDL under simulation. [Hanna97]



Introduction (cont.)



- **WAVES History**

- **Original Standard IEEE 1029.1 - 1991**
 - **Waveform And Vector Exchange Specification**
 - Focus On Exchange Between Multiple Environments
 - Focus On Automatic Test Equipment (ATE) Support
 - Focus On Documentation/Archive
 - **Subset Of VHDL Language**
 - Poor Interface For VHDL Modeling & Simulation
 - Prototype Package Implementation Inefficient, Overly Complex

Copyright © 1995-1999 SCRA

6

WAVES is the industry standard representation and exchange format for digital stimulus and response data. It provides a powerful support mechanism for concurrent engineering practices by allowing digital stimulus and response information to be freely exchanged between multiple simulation and test platforms. WAVES is defined as a syntactic subset of VHDL. Therefore, it can be simulated against a VHDL model during the design process to verify the functionality and timing of the design as it progresses. Also, when devices are fabricated, the same WAVES test vector set can be used in the electrical test process to assure that the same stimulus used during design is used during electrical test. [Flynn94], [STD97]



Introduction (cont.)



- **WAVES '97 Scope**

- Re-ballot Standard IEEE 1029.1 - 1996
 - Standard For VHDL Waveform And Vector Exchange (WAVES) to Support Design And Test Verification
 - Focus On Improved Interface Between WAVES And VHDL (1164 Support Libraries)
 - Focus On Supporting Design Verification For All Levels Of Modeling Abstraction
 - Focus On Improved Storage & Simulation Efficiency
 - Focus On Alignment With VHDL '93

Copyright © 1995-1999 SCRA

7

WAVES has been modified and improved since the initial 1991 release. A re-ballot on WAVES will result in a new IEEE standard in mid-1997. There are several improvements included in WAVES '97. The interface between WAVES and VHDL has been enhanced by using the 1164 logic values. The 1164 logic values are a base for many of the functions and libraries that WAVES requires. WAVES testbenches can be used at any level of abstraction for verification and test purposes. WAVES has also been aligned with the most recent version of the VHDL language (VHDL '93). This allows WAVES to be used with the latest features of the VHDL language. [STD97]



Module Outline



- Introduction
- **Testbench Development**
 - Design Verification Challenges
 - WAVES Concepts
 - WAVES Constructor Library and Built-Ins
 - WAVES External File
 - WAVES Test Set
 - Decoder Example
 - Algorithmic Waveform Generator Example



Testbench Development



- **Quality Testbench Is Essential For Adequate Design Verification**
- **Testbench**
 - **Methodologies Are Ad Hoc**
 - **100 Designers => 100 Approaches**
 - **Complex And Difficult To Develop**
 - **Typically On The Order Of The Model To Be Tested**
 - **Correct Model Behavior Verified “By-Observation”**

Copyright © 1995-1999 SCRA

9

A quality testbench is essential for adequate design verification. A testbench allows verification to proceed in a consistent and repeatable manner. In the past, testbenches were developed with no standards or guidelines. As a result, testbenches were often complex and difficult to produce. A hardware component under test could yield several testbenches used for verification purposes. Each testbench would require specific test inputs which were often incompatible with the other testbenches. Output results which were proven correct on one particular testbench could not be verified using the other testbenches. Therefore, the ability to repeat the same testing procedures was not often supported in previous testbenches.



Testbench Development (cont.)



- **WAVES Testbench**

- **Consistent, Structured, Standard Testbench Methodology**
- **Development Of Testbench Is Automated**
- **Correct Model Behavior Verified Automatically By Simulator**
- **Well Documented Test Set**
- **Well Documented Testing Methodology**

Copyright © 1995-1999 SCRA

10

The fundamental notion of using WAVES and VHDL together is implicit within their common VHDL foundation. However, the actual implementation of an effective WAVES integrated application is embedded within an entity called the WAVES testbench. The testbench is created to manage the insertion of test waveforms and vectors into, and the examination of results data from, the VHDL design description we wish to test. The WAVES testbench is more consistent and easier to create than previous testbenches. Automated tools and simulators can easily manipulate the WAVES testbench. [Hanna97]



Testbench Development (cont.)



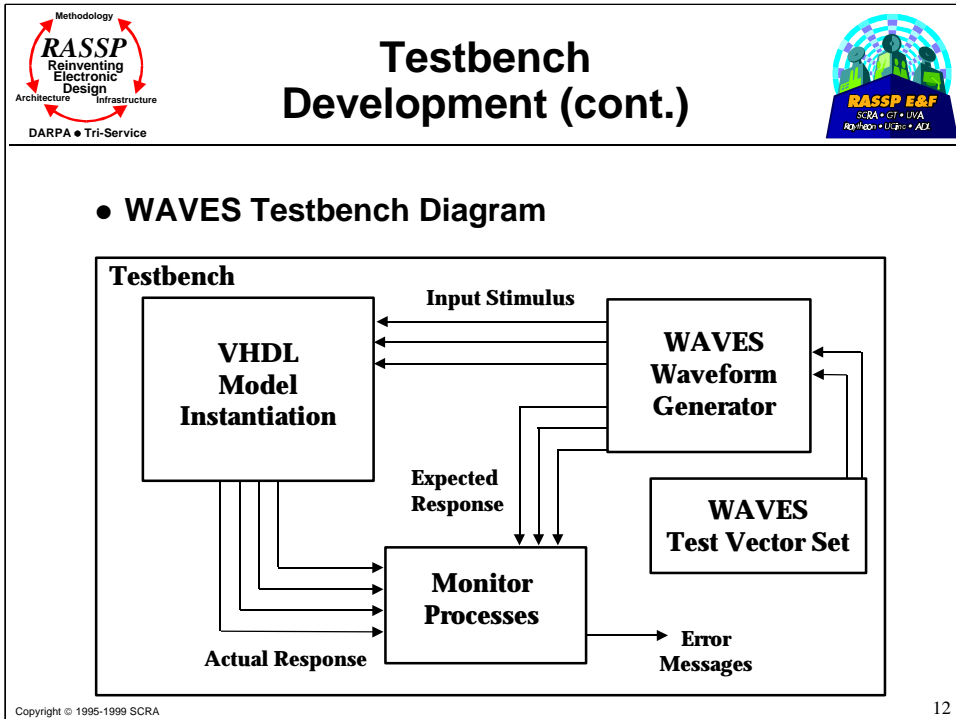
- **Details Of A WAVES Testbench**

- A VHDL Entity With No Ports
- A VHDL Architecture That Uses Structural And Behavioral VHDL Descriptions
- Architecture of Testbench Contains Instantiation Of Component Under Test, Test Vector File, Waveform Generator, Monitor Processes
- Expected Output Signals Are Declared
- Expected Outputs Compared To Actual Output Signals
- Testbench Provides Environment Similar To Real Hardware Testers (Input Stimulus, Comparison Of Output Values)
- Different Test Vector Sets Can Be Used Without Modifying Testbench

Copyright © 1995-1999 SCRA

11

The WAVES testbench is described using the standard Entity and Architecture format for a VHDL description. The testbench entity has no ports, since all testing operations occur within the testbench. The component under test is instantiated using structural VHDL. The test vector file is referenced using a VHDL File statement. The waveform generator is referenced as a VHDL Procedure within the testbench architecture. The comparison signals are declared within the architecture of the testbench. These signals are used within the monitor processes contained in the testbench architecture. The WAVES testbench is similar to real hardware testers for the reasons shown. One advantage to the WAVES testbench is that different test vector sets can be used without any modification to the testbench.



This diagram shows a complete WAVES test set. The waveform generator reads test vectors from a file and generates inputs for the component and the expected output values from that component. The component under test receives the input stimulus from the waveform generator and produces output values based on its VHDL description. These output values are then compared to the expected responses by monitor processes within the testbench. If the two responses do not match, then an error message is produced. The testbench creates a complete testing environment around the VHDL model.



Module Outline

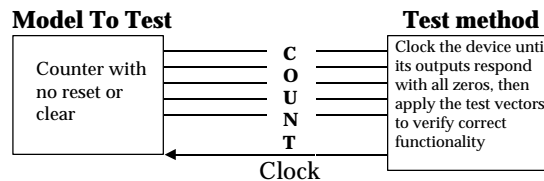


- Introduction
- Testbench Development
- **Design Verification Challenges**
 - WAVES Concepts
 - WAVES Constructor Library and Built-Ins
 - WAVES External File
 - WAVES Test Set
 - Decoder Example
 - Algorithmic Waveform Generator Example

Design Verification Challenges

• Testing Uninitialized Devices

- Device Powers up in an unknown state, has no reset (example: Counter)



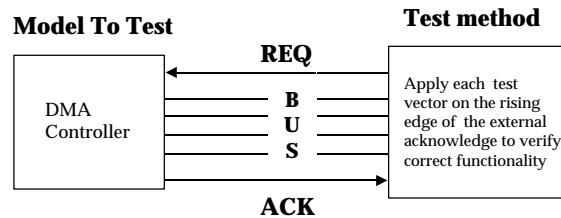
• WAVES Match Mode Test

The WAVES Match function allows the designer to perform testing of uninitialized devices. If a hardware component powers up into an unknown state and has no reset capability, it is difficult to find a valid starting point for testing. The Match functionality allows the designer to match a known value with the output of the model. This allows the designer to start testing when the behavior of the component is well established after its unpredictable initial state. In the example shown above, a counter powers up into an unknown state and has no reset. Therefore, the testing approach uses the Match functionality to establish a valid starting point to apply test vectors. In this case, the counter is clocked until it responds with all zeros on its output. The Match function compares the counter output with the desired starting value of all zeros. If they do not match, then the counter is clocked, and the output is compared again during the next time interval. If they do match, then the test vectors will be applied to the counter.

Design Verification Challenges (cont.)

- **Testing Asynchronous Devices**

- Synchronize With External Signal Before Applying Each Test Vector

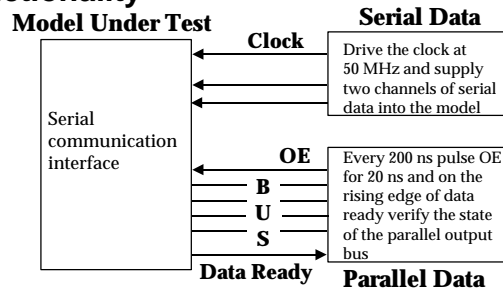


- **WAVES Handshake Mode Test**

The WAVES Handshake function allows the designer to perform testing of asynchronous devices. The Handshake functionality is designed to synchronize the application of each test vector with an asynchronous signal. In a system without a synchronous clock, the communications between component and tester is established using request and acknowledge signals. Both signals are asynchronous in nature. The tester makes a request to begin testing and the component acknowledges the request. For consistent testing, each test vector cannot be applied until the component has received the proper asynchronous acknowledgment. This insures that the test vector is applied at the proper time to perform verification. In the example shown above, the tester makes a request on the REQ line to the component. The component responds on an acknowledge (ACK) line to the tester. On the rising edge of the ACK line, the tester applies a test vector to the component. The application of the test vector has been synchronized to the asynchronous acknowledge signal.

Design Verification Challenges (cont.)

- **Testing Devices With Multiple Asynchronous Functionality**



- **Multiple WAVES Processes**

It is possible to create multiple waveform generators in WAVES which allow for multiple, simultaneous tests of a component in a system. Each waveform generator is designed to perform one type of testing for the particular component. The two waveform generators can perform their actions in parallel within the same testbench. However, each test pin can be driven only by one waveform generator. In other words, both waveform generators cannot drive the same test pins on the component. In the example shown above, the top waveform generator provides only serial data to the model under test. The bottom waveform generator drives the output enable (OE) line to allow the component to place data on the bus. When the component places the data on the bus and raises the Data Ready line, the waveform generator verifies the data on the bus. Therefore, two waveform generators are providing two separate testing operations on the same component.



Module Outline



- Introduction
- Testbench Development
- Design Verification Challenges
- **WAVES Concepts**
 - WAVES Constructor Library and Built-Ins
 - WAVES External File
 - WAVES Test Set
 - Decoder Example
 - Algorithmic Waveform Generator Example



What Is WAVES '97?



- **Essentially Sequential VHDL, PLUS**
 - Built-in Data Types, Functions, Procedures
 - External Pattern File Format
 - Standard Library Of Constructors To Support 1164 Model Verification
- **Standard, Self-contained, Unambiguous Simulatable Specification Of The Intended Behavior Of A VHDL Model/Physical Hardware**

Copyright © 1995-1999 SCRA

19

WAVES '97 is basically sequential VHDL plus new data types, functions and constructors. The data types, functions, and procedures allow waveforms to be constructed for use in testing a component. The external file format is involved in the construction of waveforms as well. The standard constructor library supports the IEEE 1164 Logic Values, which are used as the logic system throughout this module. WAVES '97 provides a self-contained and consistent method to specify and verify the intended behavior of a VHDL component description.

WAVES Concepts (cont.)

• WAVES Libraries Based On 1164 Logic Levels

U	An uninitialized state
X	An unknown forced logical state
0	A strong, forced logical low
1	A strong, forced logical high
Z	A high impedance
VV	An unknown weak logical state
L	A weak logical low
H	A weak logical high
-	Don't care

The IEEE Standard Logic Values (from IEEE Standard 1164-1993) are used throughout this module. The 1164 Logic Values are declared as an enumerated type of all legal logical values that can be used to generate events on a waveform. The WAVES libraries and constructor functions are based on the 1164 Logic Values. [Hanna97]



WAVES Concepts



- **Only A Handful Of Basic Concepts Must Be Learned**

- **Waveform**
- **Slice**
- **Events/Logic Values**
- **Frame**
- **Frame Sets/Pin Codes**
- **Frame Set Array**

Copyright © 1995-1999 SCRA

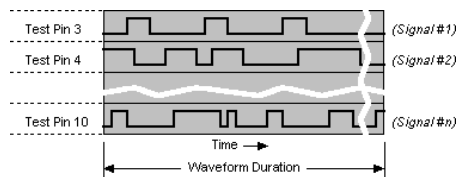
21

The most basic concepts in WAVES are the terms shown above. The waveform is constructed using the other concepts listed above. Each term will be reviewed in detail throughout this section. These terms provide the basic background to understanding the relationship between the waveform generator, test vector file, and component under test for a particular WAVES test set.

WAVES Concepts (cont.)

• Waveform

- The Set Of All Events Across All Signals For The Entire Simulation
 - Like Timing Diagram In A Data Book

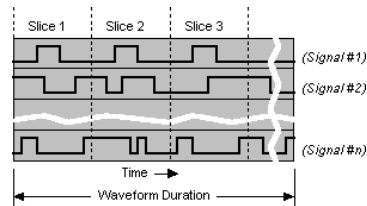


In WAVES, a waveform is simply a collection of time-dependent, logic-level transitions, or events, which have some significance in the context of the test pins of some Unit Under Test (UUT). Each series of events at a given test pin represents the time-ordered sequence of logic level changes which occur over some time duration. [Hanna97]

WAVES Concepts (cont.)

- **Slice**

- A Time Partition Of The Waveform
- Like A Tester Cycle Or Period

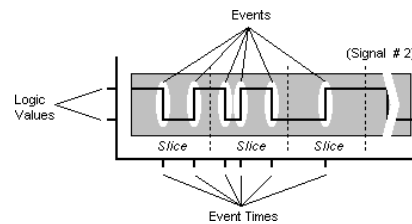


A slice is a division of a waveform and its integral signals into segments of time. The slice intervals apply uniformly across all signals in the waveform. This constraint insures a consistent view of the waveform. [Hanna97]

WAVES Concepts (cont.)

- **Events/Logic Values**

- **The Scheduling Of A Logic Value On A Signal Of The Waveform**

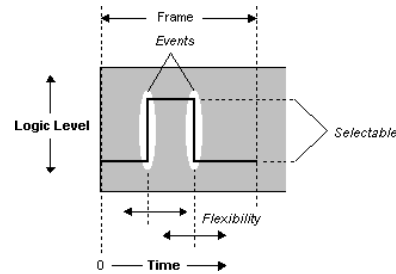


Signals in WAVES are the same as signals in VHDL. WAVES signals are used to convey sequences of logic levels for stimulating the UUT and for asserting the response that we expect from it. Signals are essentially sequences of events (not to be confused with a VHDL event). A WAVES event is a time-logic value pair. The WAVES event time is used to describe the placement, or scheduling, of the WAVES event on a signal with respect to the beginning of the current slice. The logic value component describes the discrete value of the WAVES event. Therefore, a waveform is constructed by scheduling WAVES events on the signals of a waveform on a per-slice basis. [Hanna97]

WAVES Concepts (cont.)

• Frames

- Grouping Of Events Into Common Reusable Waveform Shapes
 - The Sequence Of Events On A Given Signal For A Given Slice
 - Like A Tester Format



Frames are the fundamental building blocks of WAVES. A frame is defined as a segment of a signal between slice boundaries, which contains the events of the signal in the slice. In other words, a frame defines a particular, time-ordered sequence of logic-level transition events. However, a frame is flexible since the designer can specify the event times and logic values for the signal within a particular frame. Therefore, the designer can develop waveforms and signals by manipulating the waveform dimensions within the frames. All frames are bounded by waveform slice boundaries which specify the time references. Therefore, frames are temporally consistent across all waveform signals. [Hanna97]



WAVES Concepts (cont.)



- **Frame Sets/Pin Codes**

- **A Set Of Frames, One For Each Legal Pin Code**
 - **Defines The Events That Are Scheduled When The Given Pin Code Is Applied To A Given Signal**
 - **Pin Codes Appear In The External Pattern File**
- **Pinsets Are Used To Group Pin Codes Together**
 - **Pinsets Declared In Waveform Generator File**
 - **Used To Form A Bus of Signals Having The Same Waveform Shape**

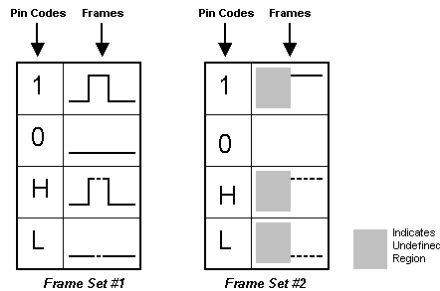
Copyright © 1995-1999 SCRA

26

A frame set is a collection of frames used to create a particular signal or group of signals. Pin codes are single-character identifiers used to select specific frames from the frame set. The same set of pin codes must be used for all frame sets for a particular test. The frame set defines a set of frames for all possible legal pattern values (WAVES pin codes) that can be used on a signal. A frame set can be generated using the various frame format functions in the WAVES constructor library. The test pins in a pinset can be described using the same frame set (in other words, the same waveform shape). [Hanna97]

WAVES Concepts (cont.)

• Frame Sets/Pin Codes (cont.)



This diagram shows two different frame sets. A sequence of pin codes from the external pattern file selects the frames needed to construct a signal in the waveform. Note that for the same pin code, two different frames are chosen from the frame sets. Therefore, even though the set of pin codes is the same for all frame sets, the constructed signals can be different from each other. [Hanna97]

WAVES 1164 Declarations

```
-- Declare the character codes that are legal in the external file.
--
constant PIN_CODES : String := "X01ZWLH-";

--
-- This is the maximum number of events that may comprise a frame.
-- A frame is a sequence of events for a given signal for a given
-- slice of the waveform. Set this constant appropriately for
-- your application.
--
constant MAX_FRAME_EVENTS : positive := 3;

--
-- Declare the logic values system that will be used to place
-- events on the waveform.
--
subtype LOGIC_VALUE is Std_Ulogic;

--
-- Declare a vector of logic values that will be used to as the
-- signal of vectors for the WAVES port list.
--
subtype WAVES_LOGIC_VECTOR is Std_Ulogic_vector;
```

This slide shows the code which declares the 1164 logic levels that are used throughout the WAVES files. The types of characters that are valid pin codes are declared in the first line. These pin codes are contained within the external test vector file.



WAVES Concepts (cont.)

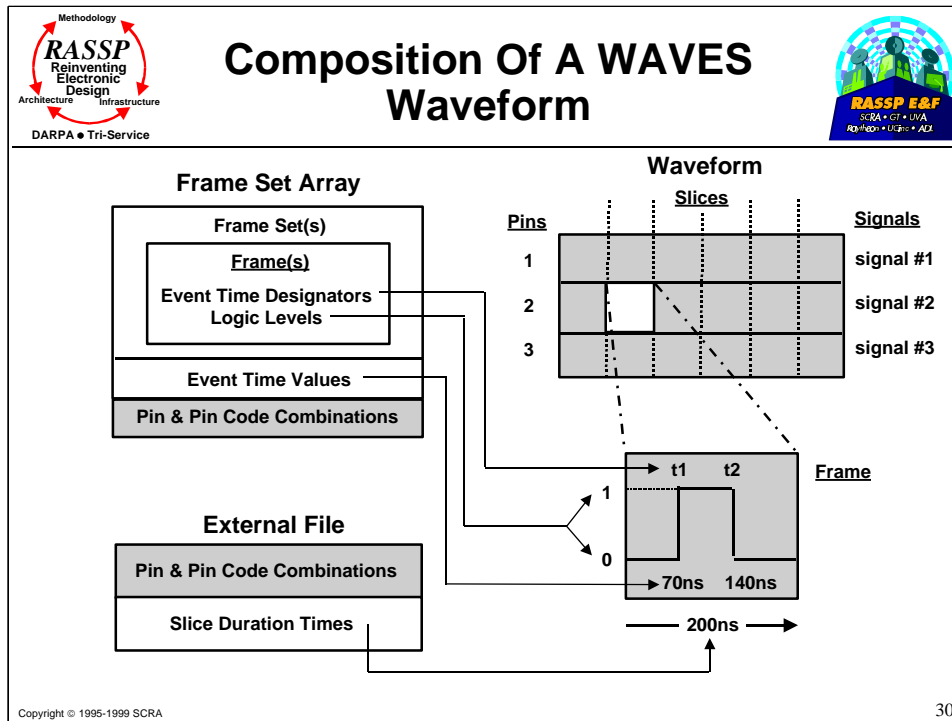


- **Frame Set Array**
 - **Associates Frame Sets With Each Signal On The Waveform**
 - **Specifies The Actual Timing For Event Scheduling**
 - **Like A Tester Timing Set**

Copyright © 1995-1999 SCRA

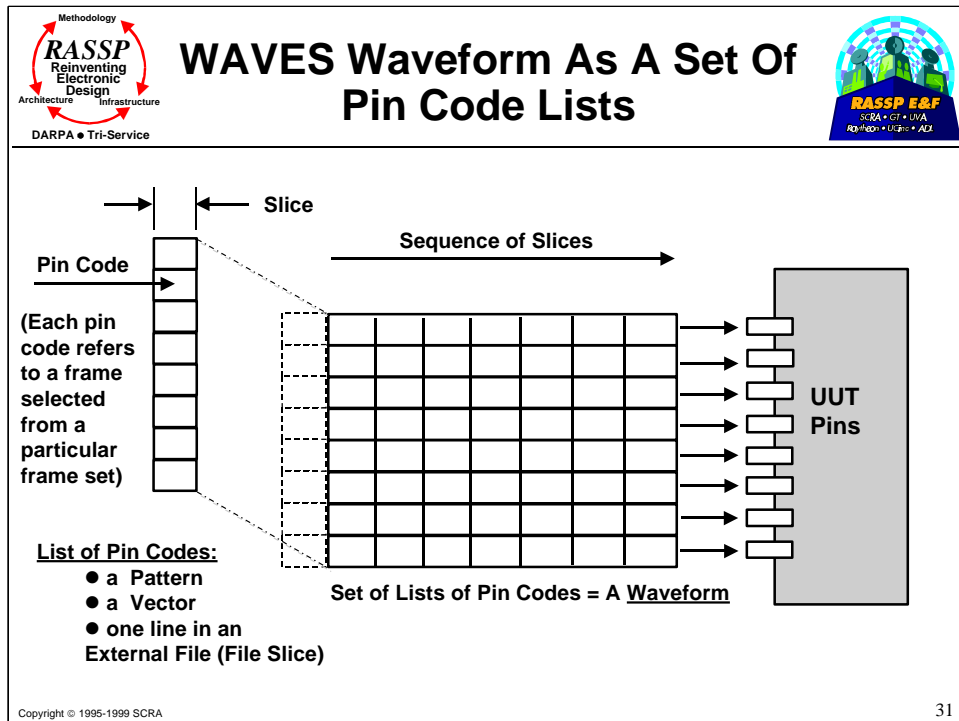
29

A frame set array serves three purposes. First, it captures all the frame sets required for the signals within a particular waveform. Second, it associates the frame sets with the particular pins of the UUT. Finally, it applies the specific time values to the event time designators within the frames. [Hanna97]



The figure shown above summarizes the contributions of the various elements to a completely defined WAVES waveform. The waveform is composed of signals associated with UUT pins. Each signal is composed of frames defined by slice duration times. Within each frame, event times define transitions between logic levels. The frame set array provides the logic level and event timing for the signals. The array includes frame sets and event time values. The information in the array is indexed by the pins and pin codes. The pin code implies the logic level as well. A pin and pin code combination in the frame set array defines a unique combination of shape, logic levels, and event times. The external file provides the slice sequencing and slice timing information, which are indexed using the same pin and pin code combinations as the frame set array.

In the figure, a particular frame is required for pin 2 of the UUT in a particular slice of time. The pin designation is 2 and the pin code is 1. Using standard logic convention, the pin code indicates that the logic level of the pulse will be a 1 during the active portion of the frame. In the frame set for pin 2, pin code 1 must designate a pulse shape, indicating two event times (t_1 and t_2). In the frame set array, two specific times with respect to the slice starting time must exist for the pin 2, pin code 1 combination (70ns and 140ns). The external file includes slice duration times for the frames associated with each pin. For the pin 2, pin code 1 combination, the slice duration is 200ns. [Hanna97]



This diagram shows the relationship between the pin codes and a constructed waveform. A slice, which is a segment of time across all signals, can be specified simply as a list of pin codes. Each pin code character is intended for a specific test pin. Recall that each test pin has an associated frame set. The pin code characters act as an index to the frame set. The pin code selects a frame in order to construct a particular test signal. A complete waveform, which consists of many sequential slices of time across all test signals, can be represented as a set of lists of pin codes, one after the other in time-slice sequence. Each pin code list specifies the frames required to define the waveform during each slice. [Hanna97]

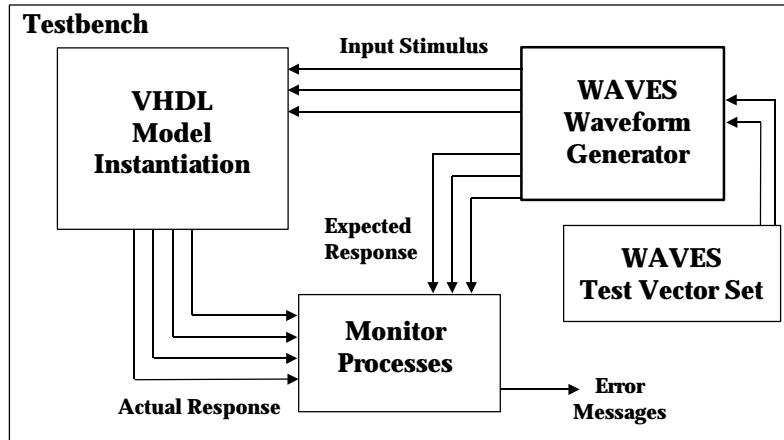


Module Outline




- Introduction
- Testbench Development
- Design Verification Challenges
- WAVES Concepts
- **WAVES Constructor Library and Built-Ins**
 - WAVES External File
 - WAVES Test Set
 - Decoder Example
 - Algorithmic Waveform Generator Example


WAVES Waveform Generator



This section of the module describes the WAVES constructor library and built-in functions which are used within the waveform generator to create the input stimulus and expected responses. The library is based on the 1164 Logic Levels shown earlier. The waveform generator generates slices of the waveform utilizing the WAVES frame and frame set building blocks and the pin codes. [Hanna97]



WAVES '97 Constructor Library

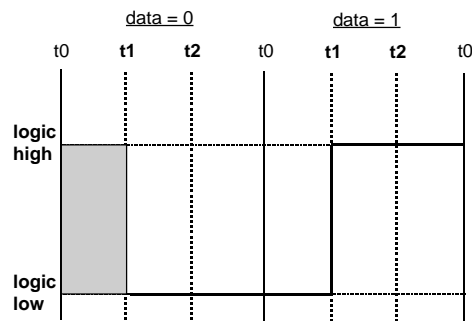


- **Provides Common Frame Shapes**
 - **Drive Format (Input Signals)**
 - **Non Return**
 - **Return High/Low**
 - **Surround By Complement**
 - **Pulse Low/High**
 - **Expected Format (Output Signals)**
 - **Window (Compare)**

Copyright © 1995-1999 SCRA34

The format of a WAVES waveform is defined using the set of functions shown above. There are two types of frame formats: drive format for input signals and expected format for the expected signals. These formats provide great flexibility in constructing waveforms. [Hanna97]

- **Non Return (user specifies t1 only)**

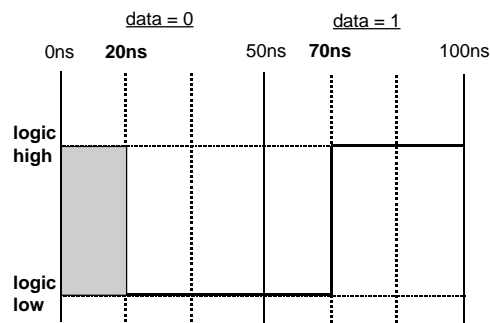


- **Function format: Non_Return (t1)**

The Non Return format is the simplest way to represent device behavior. A WAVES slice is defined from one t0 to the next t0. This example shows two slices in the Non Return format. At t0, the drive level is whatever the data of the previous slice had been. At t1, it drives the logic level of the present slice to the designated value, where it remains until the t1 of the next slice. In this example, the signal initially starts in the undefined state (the gray area). Then, at t1, since the data from the pattern file is '0', the signal is driven to a logic value '0' until the t1 in the next slice. Since the pattern data is '1' at the second t1, the signal is driven to a logic value '1'. [Hanna97]

WAVES '97 Constructor Library

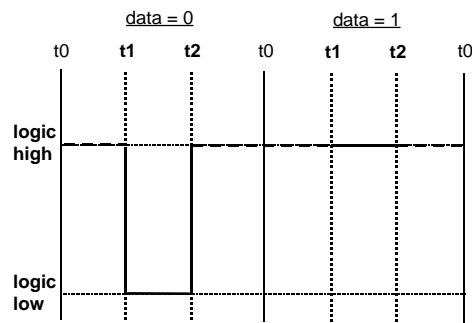
- **Non Return example**



- **Function format: Non_Return (20 ns)**

This slide shows an example of the Non Return format. In the figure shown above, each slice is 50ns in length. During the first slice (from 0ns to 50ns), the signal is initially in the undefined state. At time 20ns, the signal is driven to a logic '0' since the pattern data is a logic '0'. This value is held into the second slice (from 50ns to 100ns). At time 70ns (or 20ns from the start of the second slice), the signal is driven to a logic '1' since the pattern data is logic '1'. This value is held for the rest of the second slice as shown.

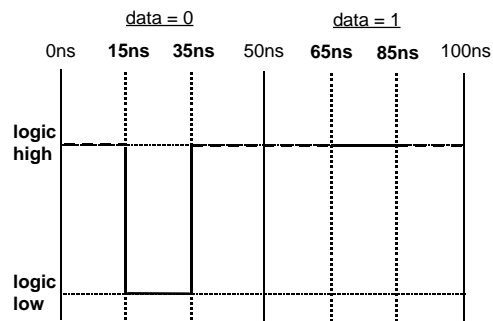
- Return High (user specifies t1 and t2)



- Function format: Return_High (t1, t2)

The Return High format drives the level high from t0 to t1 and from t2 to the following t0. Therefore, the waveform will only transition at t1 if the pattern data is a logic '0'. The dashed line on the signal represents the drive level present due to the definition of the function and not from the pattern data. In the Return High function, this default value is a logic '1'. In this example, the signal starts at the the default logic value. At t1, the pattern data is found to be a logic '0', so the signal transitions to '0' from t1 to t2. At t2, the signal returns to the default logic value. At the second t1, the pattern data is a logic '1', therefore the signal is driven to a logic '1' (shown by the solid line). At the second t2, the signal is driven back to the "default" value until the next slice. [Hanna97]

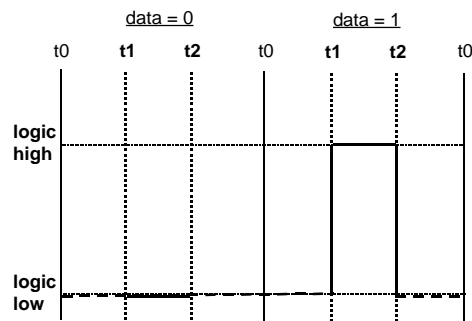
• Return High example



• Function format: Return_High (15 ns, 35 ns)

This slide shows an example of the Return High format. Each slice is 50ns in length. At 0ns, the signal is a logic '1' which is the default value for the Return High format. At 15ns, since the pattern data is a '0', the signal is driven to a logic '0'. This value is held until 35ns, when the signal returns to a logic '1'. In the second slice (from 50ns to 100ns), the signal is driven to a logic '1' throughout the entire slice since the pattern data is '1'.

- Return Low (user specifies t1 and t2)

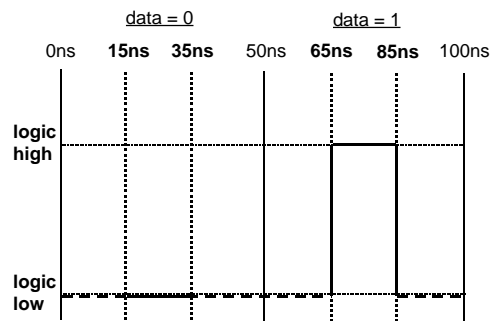


- Function format: Return_Low (t1, t2)

The Return Low format is the complement of the Return High format. It drives the level low from *t0* to *t1* and from *t2* to the following *t0*. The level is driven high from *t1* to *t2* only if the pattern data is a logic '1'. In this example, the default value is a logic '0'. Since the pattern data during the first slice is '0' the level is driven low from *t1* to *t2*. At the second *t1*, the pattern data is a logic '1'. Therefore, the level is driven at a logic '1' from *t1* to *t2* before returning to the default logic '0' value. [Hanna97]

WAVES '97 Constructor Library

• Return Low example

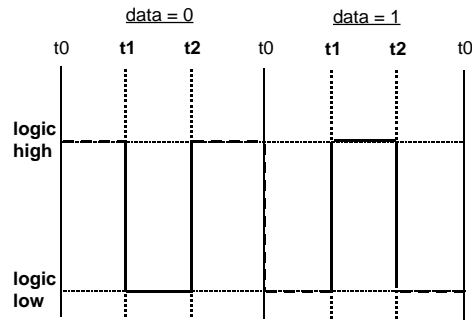


• Function format: Return_Low (15 ns, 35 ns)

This slide shows an example for the Return Low format. Each slice is defined to be 50ns in length. At 0ns, the signal is a logic '0' since the default value for the Return Low format is a logic '0'. At 15ns, the pattern data is '0' so the signal retains a value of '0' for the entire first slice. In the second slice, at 65ns, the pattern data is '1'. Therefore, the signal takes on a logic '1'. At 85ns, the signal returns to the default value of '0' for the rest of the slice.

WAVES '97 Constructor Library

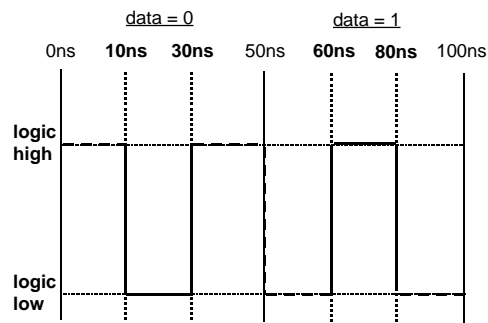
- **Surround By Complement (user specifies t1 & t2)**



- **Function format: Surround_Complement (t1, t2)**

The Surround By Complement drives the complement of the pattern data from t0 to t1 and from t2 to the following t0. The level is driven by the pattern data from t1 to t2. In the first slice, the pattern data is '0'. Therefore, the level is driven to logic '1' from t0 to t1 and t2 to the next t0. When the complement of a given frame in one slice is different than the complement value in the next slice, a transition occurs at the second t0, as shown above. This is an implied transition from the definition of the function. [Hanna97]

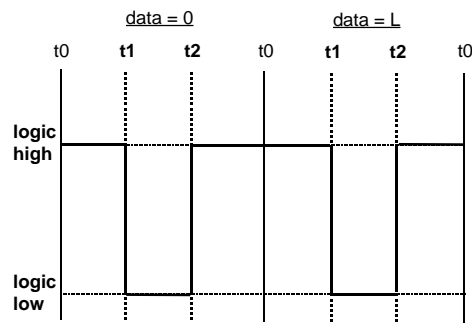
• Surround By Complement example



• Function format: Surround_Complement (10 ns, 30 ns)

This slide shows an example of the Surround By Complement format. Each slice is defined as 50ns in length. During the first slice (from 0ns to 50ns), the signal is driven to a logic '1' from 0ns to 10ns and from 30ns to 50ns. The signal is driven to a logic '1' for these intervals because the pattern data for this slice is '0'. From 10ns to 30ns, the signal is driven to a logic '0'. In the second slice (from 50ns to 100ns), the pattern data is '1'. Therefore, from 50ns to 60ns and from 80ns to 100ns, the signal is driven to a logic '0'. However, from 60ns to 80ns, the signal is driven to a logic '1' to match the pattern data.

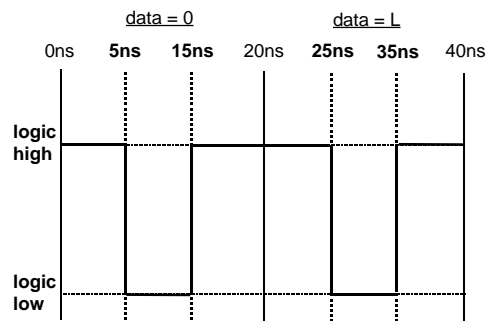
- **Pulse Low (user specifies t1 and t2)**



- **Function format: Pulse_Low (t1, t2)**

The pulse formats are used to specify periodic or regular waveforms for input signals. For example, a clock signal is efficiently described using pulse formats. In the Pulse Low format, if the vector pattern data contains either an 'L' or a '0', then the level will drive to '0' on the first edge (t_1), and stay at '0' until the second edge (t_2). In other words, it will transition from high-to-low at t_1 and from low-to-high at t_2 . There is also a Pulse Low Skew format which shifts the pulses in each slice forward in time. In this case, the user specifies the shift amount as well as t_1 and t_2 . [Hanna97]

• Pulse Low example

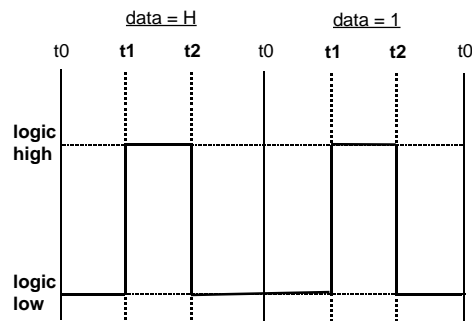


• Function format: Pulse_Low (5 ns, 15 ns)

This slide shows an example of the Pulse Low format. The slices are defined to be 20ns in length. During the first slice, the pattern data is '0' which means that there will be a 1-to-0 transition at 5ns, and a 0-to-1 transition at 15ns. This is exactly what is shown in the figure above. The signal is driven to a logic '0' from 5ns to 15ns. The times specified by the user define the rising and falling edges of the signal. In the second frame, since the pattern data is 'L', the signal will transition from 1-to-0 at 25ns, and from 0-to-1 at 35ns. This is shown in the second frame seen the figure above.

WAVES '97 Constructor Library

- **Pulse High (user specifies t1 and t2)**

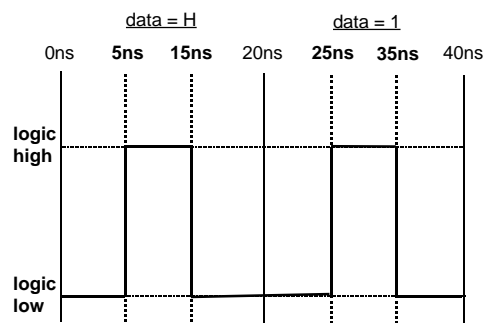


- **Function format: Pulse_High (t1, t2)**

In the Pulse High format, if the pattern data contains either an 'H' or a '1', then the level will be driven to '1' on the first edge (t1), and stay at a '1' until the second edge (t2). In other words, it will transition low-to-high at t1, and high-to-low at t2. There is also a Pulse High Skew format similar to the Pulse Low Skew format previously mentioned. [Hanna97]

WAVES '97 Constructor Library

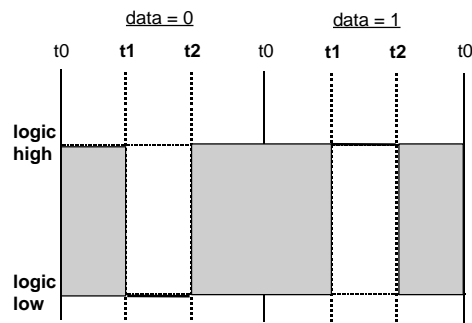
• Pulse High example



• Function format: Pulse_High (5 ns, 15 ns)

This slide shows an example of the Pulse High format. The slices are defined to be 20ns in length. During the first slice, the pattern data is 'H'. This means that a 0-to-1 transition will occur at 5ns and a 1-to-0 transition will occur at 15ns. As shown, the signal will hold a logic '1' value from 5ns to 15ns. In the second slice, the pattern data is '1'. The behavior in the second slice will be identical to the behavior in the first slice. The signal goes to a logic '1' from 25ns to 35ns as expected.

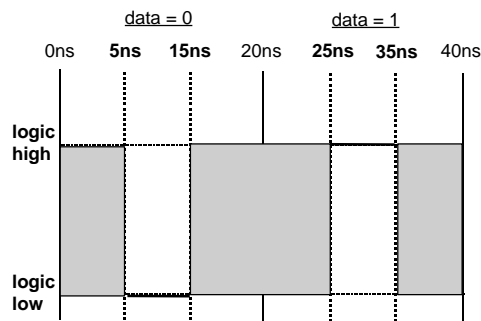
- Window (user specifies t1 and t2)



- Function format: Window (t1, t2)

The Window format is used to compare the actual UUT output with the expected output at a specified time. This format starts the valid data comparison at the t1 edge and disables the data comparison at the subsequent t2 edge. If the expected data level is not detected for the t1-to-t2 interval, or the signal changes to an invalid state, then a fail is declared. Otherwise, if the correct value is maintained for the entire interval, then the UUT output agrees with the expected output value. There is also a Window Skew format which allows the user to shift the window in each slice forward in time. [Hanna97]

• Window example



• Function format: Window (5 ns, 15 ns)

This slide shows an example of the Window format. The slices are defined to be 20ns in length. In the first slice, a comparison interval is defined from 5ns to 15ns. The pattern data value '0' and the value of the signal should match for the entire interval. In the second slice, a comparison interval from 25ns to 35ns is defined. The pattern value is '1' and the signal value should also be a logic '1' for the interval defined above.

WAVES 1164 Frame Definitions

```
function Non_Return( T1 : EVENT_TIME ) return Frame_set;  
  
function Return_Low( T1, T2 : EVENT_TIME ) return Frame_set;  
  
function Return_High( T1, T2 : EVENT_TIME ) return Frame_set;  
  
function Surround_Complement( T1, T2 : EVENT_TIME ) return Frame_set;  
  
function Pulse_Low( T1, T2 : EVENT_TIME ) return Frame_set;  
  
function Pulse_Low_Skew( T0, T1, T2 : EVENT_TIME ) return Frame_set;  
  
function Pulse_High( T1, T2 : EVENT_TIME ) return Frame_set;  
  
function Pulse_High_Skew( T0, T1, T2 : EVENT_TIME ) return Frame_set;  
  
function Window( T1, T2 : EVENT_TIME ) return Frame_Set;  
  
function Window_Skew( T0, T1, T2 : EVENT_TIME ) return Frame_Set;
```

This slides shows the definitions for all the functions used construct the WAVES frames. All of the functions require timing values as inputs, and all the functions return a frame set. In the waveform generator, one of these functions is used to construct a waveform for a particular test pin.



WAVES '97 Built-ins



- **Only A Handful of Built-in Functions**

- Apply
- Delay
- Match
- Handshake
- Read_File_Slice

- **Used In Waveform Generator File**

Copyright © 1995-1999 SCRA

50

The Apply, Delay, and Read_File_Slice are associated with the application of test vectors from the external file. Interactive waveforms can be generated using the Match and Handshake functions. [Hanna97]



WAVES '97 Built-ins



- **The Apply Procedure**
 - **Schedules Events For the Current Slice On The Waveform**
 - **Equivalent To VHDL Signal Assignment**
- **The Delay Procedure**
 - **Suspends The Execution Of the WAVES Process For The Given Time Duration**
 - **Equivalent To VHDL “wait for”**

Copyright © 1995-1999 SCRA

51

The Apply and Delay procedures are used in the waveform generator. The Apply procedure takes the pattern data read from the external file and schedules events on the test pins for the current slice. The timing of the events is defined by the frame format functions shown earlier. The Delay procedure uses the slice timing information to suspend the WAVES process for a certain duration. After applying a test vector, the simulation will have to advance time to finish the current slice. After the Delay procedure is executed, the current time is incremented by the value of the delay time. Then, a new test vector could be applied to the current slice. [STD97]



WAVES '97 Built-ins



- **The Match Procedure**

- Initiates A *Match* Request To The Testbench
 - Provides Coarse (Value Level) Synchronization

- **The Handshake Procedure**

- Initiates A *Handshake* Request To The Testbench
 - Provides Event Level Synchronization
 - Suspends Execution Of The WAVES Process Until The *Handshake* Request Is Satisfied

Copyright © 1995-1999 SCRA

52

The Match procedure is used to initiate a Matching operation in the WAVES test set. It could be used to test an uninitialized device that has no reset capability. The Match procedure specifies which test pins are compared to a known initial value. The Match operation is terminated when the value on the test pins matches the initial value. The Handshake procedure is used to achieve synchronization between asynchronous signals and the application of a test vector. This procedure specifies which signals are used to perform the Handshaking operation. This functionality will not apply a new test vector to the UUT until it is synchronized through the Handshaking process.



WAVES '97 Built-ins



- **The Read_File_Slice Procedure**
 - Reads A Vector (Slice) From A WAVES External Pattern File
 - Predefined File Format

Copyright © 1995-1999 SCRA

53

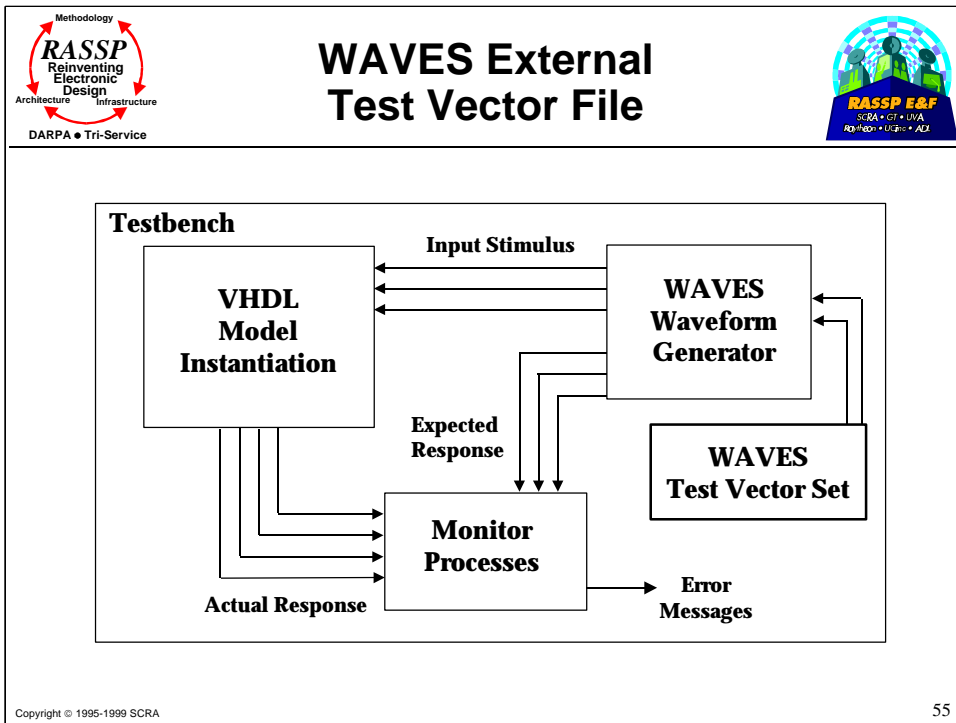
The Read_File_Slice is used to read data from the external file and store it in a temporary variable within the waveform generator. The Apply and Delay functions will use certain parts of the stored test vector to perform their operations. The predefined file format will be discussed in the next section.



Module Outline



- Introduction
- Testbench Development
- Design Verification Challenges
- WAVES Concepts
- WAVES Constructor Library and Built-Ins
- **WAVES External File**
- WAVES Test Set
- Decoder Example
- Algorithmic Waveform Generator Example



55

This sections describes the format for the WAVES test vector file. The data within the file must strictly adhere to the WAVES format for the waveform generator to correctly read and process the file slices. The waveform generator reads pin codes and slice information one line at a time from the external file, and constructs a corresponding slice of the waveform. [Hanna97]



WAVES '97 External File



- **Separates Test Pattern Data From Functional WAVES Code**
- **Organized Into File Slices**
 - One "Pin Code" Character Per Device Pin
- **Patterns May Be Specified As Binary Bit Level Patterns Or In Hexadecimal**
 - May Freely Mix Binary And Hexadecimal Fields
- **Could Algorithmically Generate Test Vectors Within Waveform Generator File (an external pattern file would not be required in this case)**

Copyright © 1995-1999 SCRA

56

An external file contains pin codes and slice timing information necessary to build a complete waveform. Each line in an external file represents a time slice in a waveform across all signals. Each pin code for a particular file slice is intended for a particular test pin. The patterns can be binary or hexadecimal in format as shown in the next few slides. [Hanna97]

WAVES External File (cont.)

- **File Slices Contain Two “Token” Types**

- **Data Tokens Specify Pattern Data**
 - **Binary Pin Code Tokens**
 - **Hexadecimal Tokens**
- **Control Tokens Specify Processing**
 - **Timing Specifier**
 - **Skip Command**
 - **Repeat Command**
 - **Hexadecimal Template**

File slices contain two types of tokens: data tokens and control tokens. The data tokens are specified in binary or hexadecimal format. The control tokens specify slice timing and other operations. Many of the specific types of tokens are identified in the external file by special characters, which are described later in this section.



WAVES External File (cont.)



- **File Slice Structure**

- Any Mixture Of Data And Control Tokens
- Spaces And Tabs Are Permitted
- Data Tokens Must Be Terminated By Space Or Tab
- Slice May Span Several Lines Of The File
- Semicolon ';' Is The Slice Terminator

Copyright © 1995-1999 SCRA

58

The format of each file slice must obey the requirements shown above. There can be any number of data tokens and control tokens in a given file slice. The data tokens are terminated by a space or tab. A file slice is not limited to one line in the file. It could span several lines in the file as needed for a particular test vector set. The semicolon character is used to terminate a file slice within the external file. [Hanna97], [STD97]

- **Special Characters**

- **'%' Introduces A Comment**
 - **Comments Span To The End Of The Line**
- **'=' Introduces A Skip Command**
 - **Applies The Next Data Token Beginning At Column *n* (example: =32 1010)**
- **'+' Introduces A Repeat Command**
 - **Repeat Vector *n* Times (example: +200)**

The special characters in the external file are used to specify the types of tokens and operations included in a particular file slice. The '%' character is used to declare a comment line in the external file. The '=' character is used to introduce a skip command. A skip command allows the data tokens to be applied to specific columns in the file slice. Each column in a line of data tokens is assigned to a specific test pin. The first test pin (test pin #1) is the leftmost data token in the slice. The skip command could be used when most of the data tokens in the current file slice are the same as the tokens in the previous file slice. In the example shown above, the "=32" term indicates which pin number the tokens "1010" should be applied to. Therefore, the data tokens for pins 1 through 31 remain the same as the previous file slice, while pins 32 through 35 receive the data tokens shown above. Pin #32 receives '1', Pin #33 receives '0', and so forth. The '+' character is used to signal a repeat command. If a repeat command is included in a file slice, then that file slice will be applied the number of times specified. In the example shown, the "+200" means that the current file slice will be applied 200 times. After 200 applications, the next file slice is read and processed. [Hanna97], [STD97]

WAVES External File Example

```
% In this example, the data tokens are in binary format only.
% The control tokens are timing specifiers which indicate the slice
% duration.
00010010 : 20 ns;
% data tokens => 00010010      control token => : 20ns
01110111 : 15 ns;
01111100 : 10 ns;
```

```
% In this example, the skip and repeat commands are shown below.
% The first test vector is in binary format and has a 20ns duration:
00010010 : 20 ns;
% The second test vector uses the skip command to apply the data
% tokens at column #5.  In this case, the resultant vector is
% 00010111 with a 20ns duration.
=5 0111 ;
% The third test vector will be repeated 20 times with a 20ns
% duration during each application.
+20 01111100 ;
```

This slide shows some examples which use data tokens, control tokens, and the special characters. The first example shows test vectors for a device with 8 test pins. As a result, there are 8 data tokens shown in binary format. The timing specifier indicates that the duration of the first slice is 20ns, the second slice is 15ns, and the third slice is 10ns.

The second example shows the skip and repeat commands. Again, the UUT has 8 test pins, so each file slice must produce 8 data tokens. The time duration of 20ns is the same for all vectors in this example. The first vector declares a timing specifier of 20ns. Since the other two vectors do not have a timing specifier, the previous slice duration of 20ns must be used. The second vector shows 4 tokens which we want to apply to pins 5 through 8 only. The data tokens for pins 1 through 4 are the same as the first vector. The third vector shows the repeat command as shown.

- **Special Characters (cont.)**

- **‘^’ Introduces Standard Hex Field**
 - example: ^A5 => 10100101
- **‘#’ Introduces User Defined Hex Template**
 - Used To Define The Binary Pin Code String Generated By The Hex Field (example: #LH)
- **‘~’ Introduces User Hex Field**
 - example: ~A5 => HLHLLHLH

The ‘^’ character is used to signify data tokens in the standard hexadecimal format. In this example “^A5” corresponds to the binary data tokens of “10100101”. The “#” character is used to create a user-defined hexadecimal template. The user defines which pin codes are used to translate a user-defined hexadecimal field into a pin code bit pattern. The first pin code after the # sign is used as the “zero” character. The second pin code is used as the “one” character. When a user-defined template is used, the hexadecimal data field is translated into a series of data tokens. The data tokens must be one of the pin codes specified in the template. In the example shown, the pin code ‘L’ will be used like a ‘0’ in binary format. The pin code ‘H’ will be used like a ‘1’ in binary format. After a user-defined hexadecimal template is declared, the ‘~’ character must be used to signify a user-defined hexadecimal data field. In the example, “~A5” signifies a user-defined hexadecimal template. If the template “#LH” was declared, then “~A5” would correspond to “HLHLLHLH”. Note that instead of using ones and zeros, we used the pin codes ‘H’ and ‘L’ to represent a translation of the hexadecimal data. [Hanna97], [STD97]

WAVES External File Example (cont.)

```
% In this example, the data tokens are in standard hex format only.
% The control tokens are timing specifiers which indicate the slice
% duration.
^12 : 20 ns;
^77 : 15 ns;
^7C : 10 ns;
```

```
% In this example, the data tokens are in user-defined hex format.
% The control tokens are timing specifiers which indicate the slice
% duration.
#LH ~12 : 20 ns;
% data tokens => LLLHLLHL      control tokens => #LH, : 20ns
~77 : 15 ns;
% data tokens => LHHHLHHH      control token => : 15ns
% This vector uses the standard hex format
^77 : 15 ns;
% data tokens => 01110111      control token => : 15ns
```

This slides shows some example for the hexadecimal format. In the first example, the data tokens are in standard hexadecimal format with time duration specified. In the second example, a user-defined hexadecimal template has been declared using the 'L' and 'H' pin codes. The data tokens and control tokens for each vector are shown.

- **Special Characters (cont.)**

- **‘:’ Introduces A Timing Specifier**
 - **May Be An Integer Or A Time**
 - **Time Indicates Duration Of Vector (example: 20 ns)**
 - **Integer Indicates Timing Set Selection**
- **‘;’ Indicates The End Of The File Slice**

The colon character ‘:’ is used to specify timing information. If a period of time is specified after the ‘:’ character, then the slice duration is specified. If an integer is specified after the ‘:’ character, then the integer indicates which frame set array in the waveform generator is to be used to apply the current file slice. In many applications, multiple timing sets in the waveform generator are required to create the necessary waveforms. For example, a component may have bi-directional pins. Therefore, one timing set would be used for input mode on the pin, while the other would be used for output mode on the pin. The integer after the colon serves as an index to the proper timing set for a particular file slice. The semicolon character ‘;’ is used to terminate a file slice. [Hanna97], [STD97]

WAVES External File Example (cont.)

```
% In this example, the data tokens are in binary format only.  
% The control tokens are timing specifiers which indicate the timing  
% set selection.  
00010010 : 2;  
% data tokens => 00010010      control token => : 2  
% this vector is using timing set 2  
01110111 : 1;  
% this vector is using timing set 1  
01111100 : 2;
```

In this example, the timing command using integers is shown. The integer selects which timing set in the waveform generator that is used for each file slice. In this example, there are two timing sets in the waveform generator, indexed as '1' or '2'.

- **Hexadecimal Data Field Format**

- **Field Width Specifier Allowed (^XX:n)**
 - **Narrower Than Hex Field Specified**
 - **Truncate Left Most Characters**
 - **example: ^FF08:14 => 11111100001000**
 - **Wider Than Hex Field Specified**
 - **Fill With “Zeros” On The Left**
 - **example: ^AA:16**
 - **Generates: 0000000010101010**

When a ‘:’ character is included in a hexadecimal data field, the width of the pin code token has been specified. This allows the user to truncate characters from the tokens or to add characters to the tokens.

Normally, a single hexadecimal character are converted into four pin codes. If the specified width is less than the normal hexadecimal conversion, then the leftmost characters are removed. In the example shown, ^FF08:14 signifies that the resultant pin codes from this hexadecimal data is to be 14 characters wide. Normally, ^FF08 would result in 16 data tokens. In this example, the two leftmost characters are removed, which results in a translation from “^FF08” to “11111100001000”. If the specified width is larger than the normal hexadecimal conversion, then extra zeros are added to the data tokens. In the second example, ^AA:16 requires 16 data tokens after conversion. However, the hexadecimal number AA is only 8 tokens wide. Therefore, 8 extra zeros are added to produce the data token string of “0000000010101010”. [STD97]

- **Hexadecimal Data Field Format (cont.)**

- **Rules For Field Expansion**
 - **Hexadecimal Fields Generate Contiguous Character Patterns (example: 32 Bits Wide Starting In Column 10)**
 - **Total Generated Width Of The Entire Pattern May Not Exceed The Number Of Device Pins**
- **Applies To Both Standard Hex Fields And User Defined Hex Fields**

There are two rules which all standard and user-defined hexadecimal data fields must follow. The first rule is that hexadecimal fields generate contiguous pin codes. Therefore, when a hexadecimal field is translated into pin codes, then all the resultant pin codes must be next to each other within the file slice. In the example, if we specified that a converted 8-character hexadecimal field would start at column 10. This means that 32 pin codes would be placed from column 10 to column 41. The other rule sets a limit on the maximum width of pin codes in a file slice. If the user specifies a width for a hexadecimal translation, then this width must be less than the number of test pins on the UUT. If the specified width is greater than the number of test pins, an error will result. [STD97]



Module Outline



- Introduction
- Testbench Development
- Design Verification Challenges
- WAVES Concepts
- WAVES Constructor Library and Built-Ins
- WAVES External File
- **WAVES Test Set**
- Decoder Example
- Algorithmic Waveform Generator Example

WAVES Test Set

- **Structure Of A WAVES 1164 Test Set**
 - **UUT Test Pins Package**
 - **Enumerated Type Declaration (Test_pins)**
 - **Waveform Generator Package**
 - **Contains One Or More WAVES Waveform Generator Procedures (WAVES Processes)**
 - **Testbench File**
 - **Optional External Pattern File**
 - **Predefined WAVES 1164 Constructor Library And WAVES Standard Packages**

This slides shows the structure of a complete WAVES test set. It includes a test pin file, waveform generator, testbench, external pattern file, and the WAVES 1164 libraries and standard packages.



WAVES Test Set (cont.)



- **The WAVES Header File**

- **Data Set Identification**
 - **Author, Revision Level, Device Name, and so forth**
- **WAVES File Identification**
 - **Compilation Order**
 - **File Dependencies**
- **External File Identification**
 - **Identifies Test Vector File**
- **Waveform Identification**
 - **Specifies The Waveform Generator Procedures**

Copyright © 1995-1999 SCRA

69

The WAVES header file is used for documentation purposes and does not need to be compiled with the rest of the WAVES test set. The WAVES header file identifies the WAVES data set and describes how the data set is to be assembled. It also identifies the external file which contains the pin codes to be used as part of the data set and specifies the name of the waveform generator procedures. WAVES was created as an exchange specification for waveforms and test vectors between different organizations. Therefore, the documentation and identification information in the header file is important for understanding various test sets. [Hanna97], [STD97]

WAVES Header File Example

```
-- ***** Generic Header File

-- Data Set Identification Information
TITLE      WAVES Test Set for Component X
DEVICE_ID   component_x
DATE        Wed July 16 16:03:05 1997
ORIGIN      Component X Design Team
AUTHOR      The Company of X, Y, and Z
OTHER       comments relating to the data set

-- Data Set Construction Information
VHDL_FILENAME component_x.vhd                WORK
WAVES_FILENAME test_pins.vhd                WORK
library     IEEE;
use          IEEE.WAVES_1164_Declarations.all;
use          IEEE.WAVES_Interface.all;
use          WORK.UUT_Test_pins.all;
WAVES_UNIT   WAVES_OBJECTS                  WORK
WAVES_FILENAME component_x_wgp.vhd          WORK
VHDL_FILENAME component_x_tstbench.vhd      WORK
EXTERNAL_FILENAME component_x_vect.txt      VECTORS
WGEN_PROCEDURE WORK.wgp_component_x.waveform
```

This example shows a generic header file for a “component_x”. The Data set identification section shows the author, device name, and other information. If this data set was modified in the future by another design team, then further information may be included in this section. The WAVES files are identified in the second section. The name of the component VHDL description is shown along with the working library it is compiled into. The file name for the test pin file is shown next. The IEEE library is declared and various WAVES packages are identified. These libraries and packages are used when the WAVES test set is compiled. The filenames for the waveform generator, the testbench, and the external file are shown next. Finally, the name of the waveform generator procedure is shown at the bottom. The library and package where the waveform generator procedure is declared are shown above.

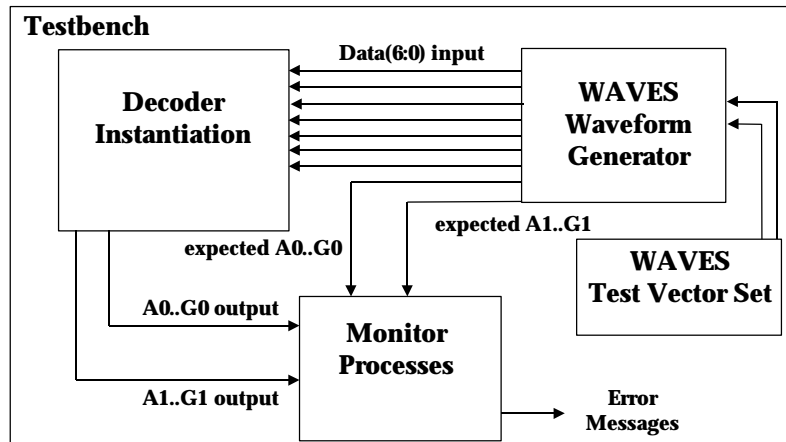


Module Outline



- Introduction
- Testbench Development
- Design Verification Challenges
- WAVES Concepts
- WAVES Constructor Library and Built-Ins
- WAVES External File
- WAVES Test Set
- **Decoder Example**
- Algorithmic Waveform Generator Example

Decoder Example



This WAVES example uses a decoder circuit as the component under test. This decoder is designed to accept a 7-bit binary coded value as the input. The outputs of the decoder are intended to drive a two digit, 7-segment LED display. The LED display should show the correct decimal display for the binary input value. The figure above summarizes the WAVES test set for the decoder. The VHDL for the entire WAVES test set is shown on the next several slides. It is important to recognize what changes must be made to each WAVES file in order to produce a simulatable WAVES test set.



Decoder Example



- **This WAVES Example Consists Of 6 Files:**

- Header File
- Behavioral VHDL Description Of Decoder
- Test Pins File
- External Test Vector File
- Waveform Generator File
- Testbench File

Header File: Decoder Example

```
-- ***** Header File for Entity: display_driver
-- Data Set Identification Information
TITLE      WAVES Decoder Test set example
DEVICE_ID  display_driver
DATE       Tue Mar 17 16:25:14 1998
ORIGIN     RASSP E&F and UVA
AUTHOR     UVA EE Dept
DATE       Tue Mar 17 16:25:14 1998
OTHER      This example demonstrates a complete WAVES test
OTHER      set for a decoder device.
-- Data Set Construction Information
VHDL_FILENAME  bcd_decoder.vhd                WORK
WAVES_FILENAME bcd_decoder_pins.vhd           WORK
library
use            IEEE;
use            IEEE.WAVES_1164_Declarations.all;
use            IEEE.WAVES_Interface.all;
use            WORK.UUT_Test_pins.all;
WAVES_UNIT     WAVES_OBJECTS                  WORK
WAVES_FILENAME bcd_decoder_wgen.vhd           WORK
VHDL_FILENAME  bcd_decoder_tstbench.vhd       WORK
EXTERNAL_FILENAME  bcd_decoder_vectors.txt    VECTORS
WAVEFORM_GENERATOR_PROCEDURE  WORK.wgp_display_driver.waveform
```

As stated before, the WAVES header file provides documentation about a particular WAVES test set. The data set identification section gives the name of the component under test, date, and origin information. The data set construction information shows the various filenames in the WAVES test set. The WAVES library dependencies are shown, as well as the filenames for the component, test pin file, testbench, waveform generator, and external pattern file.

Decoder VHDL Description

```
ENTITY display_driver IS
  PORT (data : IN std_logic_vector(6 DOWNTO 0);-- binary data input
        A0  : OUT std_logic;                -- segments for digit 0
        B0  : OUT std_logic;
        C0  : OUT std_logic;
        D0  : OUT std_logic;
        E0  : OUT std_logic;
        F0  : OUT std_logic;
        G0  : OUT std_logic;
        A1  : OUT std_logic;                -- segments for digit 1
        B1  : OUT std_logic;
        C1  : OUT std_logic;
        D1  : OUT std_logic;
        E1  : OUT std_logic;
        F1  : OUT std_logic;
        G1  : OUT std_logic);
END display_driver;
```

This is a standard entity VHDL description for the decoder. It has a total of 21 ports. The 7-bit bus data is the input signal to this circuit. The fourteen output signals are the result of the decoding operation. The ports A0 through G0 are meant to drive the lower digit in the 7-segment display. The ports A1 through G1 are meant to drive the upper digit in the 7-segment display.

Decoder VHDL Description (cont.)

```
ARCHITECTURE default OF display_driver IS

    SIGNAL display1 : std_logic_vector(6 DOWNTO 0);
    SIGNAL display0 : std_logic_vector(6 DOWNTO 0);

BEGIN

    decode_process : PROCESS(data)
        VARIABLE int_data : INTEGER := 0;
        VARIABLE tens_place : INTEGER := 0;
        VARIABLE ones_place : INTEGER := 0;
    BEGIN
        -- Convert the input data to an integer
        int_data := to_integer(data);
```

This is the behavioral architecture for the decoder. The main process is sensitive only to the data signal. Three integer variables are declared in order to process the binary input. Two signals are declared to store the output values for each digit of the 7-segment display.

Decoder VHDL Description (cont.)

```
-- Next, determine the values for the ten's place digit and
-- the one's place digit
IF (int_data > 99) THEN
    -- Error - assign both values = -1
    ones_place := -1;
    tens_place := -1;
ELSIF (int_data >= 90) THEN
    ones_place := int_data - 90;
    tens_place := 9;
ELSIF (int_data >= 80) THEN
    ones_place := int_data - 80;
    tens_place := 8;
ELSIF (int_data >= 70) THEN
    ones_place := int_data - 70;
    tens_place := 7;
ELSIF (int_data >= 60) THEN
    ones_place := int_data - 60;
    tens_place := 6;
```

The binary input is converted into an integer. This integer representation is used to select the values for the variables ones_place and tens_places.

Decoder VHDL Description (cont.)

```
ELSIF (int_data >= 50) THEN
    ones_place := int_data - 50;
    tens_place := 5;
ELSIF (int_data >= 40) THEN
    ones_place := int_data - 40;
    tens_place := 4;
ELSIF (int_data >= 30) THEN
    ones_place := int_data - 30;
    tens_place := 3;
ELSIF (int_data >= 20) THEN
    ones_place := int_data - 20;
    tens_place := 2;
ELSIF (int_data >= 10) THEN
    ones_place := int_data - 10;
    tens_place := 1;
ELSE
    ones_place := int_data;
    tens_place := 0;
END IF;
```

Decoder VHDL Description (cont.)

```
-- Finally, set the values of the output variables that drive
-- the 7-segment displays according to the values of
-- tens_place and ones_place
--
--                                     abcdefg
CASE ones_place IS
    WHEN 0      => display0 <= "1111110"; -- zero
    WHEN 1      => display0 <= "0110000"; -- one
    WHEN 2      => display0 <= "1101101"; -- two
    WHEN 3      => display0 <= "1111001"; -- three
    WHEN 4      => display0 <= "0110011"; -- four
    WHEN 5      => display0 <= "1011011"; -- five
    WHEN 6      => display0 <= "1011111"; -- six
    WHEN 7      => display0 <= "1110000"; -- seven
    WHEN 8      => display0 <= "1111111"; -- eight
    WHEN 9      => display0 <= "1110011"; -- nine
    WHEN OTHERS => display0 <= "1001111"; -- error
END CASE;
```

The variables `ones_place` is used to control the lower digit in the 7-segment display. The output ports A0, B0, C0, D0, E0, F0, G0 are set using the signal `display0`.

Decoder VHDL Description (cont.)

```
CASE tens_place IS
    WHEN 0      => display1 <= "1111110"; -- zero
    WHEN 1      => display1 <= "0110000"; -- one
    WHEN 2      => display1 <= "1101101"; -- two
    WHEN 3      => display1 <= "1111001"; -- three
    WHEN 4      => display1 <= "0110011"; -- four
    WHEN 5      => display1 <= "1011011"; -- five
    WHEN 6      => display1 <= "1011111"; -- six
    WHEN 7      => display1 <= "1110000"; -- seven
    WHEN 8      => display1 <= "1111111"; -- eight
    WHEN 9      => display1 <= "1110011"; -- nine
    WHEN OTHERS => display1 <= "1001111"; -- error
END CASE;
END PROCESS;
```

The variables `tens_place` is used to control the upper digit in the 7-segment display. The outputs A1, B1, C1, D1, E1, F1, G1 are set using the signal `display1`.

Decoder VHDL Description (cont.)

```
a0 <= display0(6);  
b0 <= display0(5);  
c0 <= display0(4);  
d0 <= display0(3);  
e0 <= display0(2);  
f0 <= display0(1);  
g0 <= display0(0);  
  
a1 <= display1(6);  
b1 <= display1(5);  
c1 <= display1(4);  
d1 <= display1(3);  
e1 <= display1(2);  
f1 <= display1(1);  
g1 <= display1(0);  
  
END default;
```

The output of the CASE statement selections are mapped to the output signals. The variable ones_place has been used to set signals A0 through G0. The variable tens_place has been used to set signals A1 through G1.

Decoder Test Pin File

```
-- ***** This File Was Automatically Generated *****
-- ***** By The WAVES96-VHDL Tool Set *****
-- ***** Generated for Entity: display_driver *****
-- ***** This File Was Generated on: Fri Jan 30 16:43:40 1998 *****
--
--
PACKAGE uut_test_pins IS
TYPE test_pins IS (data_6, data_5, data_4, data_3, data_2, data_1,
                   data_0, A0, B0, C0, D0, E0, F0, G0, A1, B1, C1,
                   D1, E1, F1, G1);
END uut_test_pins;
```

- A new TYPE is declared for the external pins of the decoder.

The test pin file can be automatically generated by the WAVES tool set. No changes need to be made here if the WAVES tools are used. The test pins are declared as a new type, and a new VHDL package “uut_test_pins” is created. The values of test_pins are the names of pins used to test or exercise the UUT.

Decoder Test Vector File (excerpt)

```
% DATA  ABCDEFG  ABCDEFG
%          1111111  0000000
0000000  1111110  1111110 : 500 ns ;
0000001  1111110  0110000 : 500 ns ;
0000010  1111110  1101101 : 500 ns ;
0000011  1111110  1111001 : 500 ns ;
0000100  1111110  0110011 : 500 ns ;
0000101  1111110  1011011 : 500 ns ;
0000110  1111110  1011111 : 500 ns ;
0000111  1111110  1110000 : 500 ns ;
0001000  1111110  1111111 : 500 ns ;
0001001  1111110  1110011 : 500 ns ;
0001010  0110000  1111110 : 500 ns ;
0001011  0110000  0110000 : 500 ns ;
0001100  0110000  1101101 : 500 ns ;
0001101  0110000  1111001 : 500 ns ;
0001110  0110000  0110011 : 500 ns ;
0001111  0110000  1011011 : 500 ns ;
```

- Each file slice contains data tokens for every test pin on the decoder. The duration for each slice is 500 ns.

This is an excerpt from the test vector file for the decoder. The lines beginning with “%” are comments. Each file slice contains 21 pin code characters. The first 7 pin codes are used to drive the input bus data. The next 7 pin codes represent the expected outputs on signals A1 through F1. The last 7 pin codes are used to drive the expected output signals for A0 through F0. The external test vector file **cannot** be created by the WAVES tools, so the designer must create this file.

Decoder Waveform Generator File

```
--
-- ***** This File Was Automatically Generated *****
-- ***** By The WAVES96-VHDL Tool Set *****
-- ***** Generated for VHDL entity: *****
-- ***** display_driver *****
-- ***** Generation date and time: *****
-- ***** Fri Jan 30 16:43:40 1998 *****
--

use STD.TEXTIO.all;
library IEEE;
use IEEE.WAVES_1164_Frames.all;
use IEEE.WAVES_1164_Declarations.all;
use IEEE.WAVES_Interface.all;
use WORK.WAVES_Objects.all;
use WORK.UUT_Test_Pins.all;

package WGP_display_driver is

    procedure WAVEFORM( signal WPL : inout WAVES_PORT_LIST );

end WGP_display_driver;
```

The waveform generator can be automatically generated using the WAVES tool set. However, it **must be modified** by the designer before it can be simulated. The designer must insure that the library references for the waveform generator are correct. In this example, the WAVES library is used for the standard WAVES packages. The package “uut_test_pins” is from the test pin file and was compiled to the “work” directory. A package “WGP_display_driver” is declared which contains the actual waveform generator. A signal “WPL” of type Waves_Port_List is used to communicate the waveform information with the UUT during simulation. [Hanna97]

Decoder Waveform Generator File (cont.)

```
package body WGP_display_driver is

  procedure WAVEFORM( signal WPL : inout WAVES_PORT_LIST ) is

    file VECTOR_FILE : text open READ_MODE is "bcd_decoder_vectors.txt";

    variable VECTOR : FILE_SLICE := NEW_FILE_SLICE;

    constant data : PINSET := data_6 + data_5 + data_4 + data_3 +
      data_2 + data_1 + data_0;

    constant OUT_PINS : PINSET := A0 + B0 + C0 + D0 + E0 + F0 +
      G0 + A1 + B1 + C1 + D1 + E1 + F1 + G1;

    constant DISP1 : PINSET := A1 + B1 + C1 + D1 + E1 + F1 + G1;
    constant DISP0 : PINSET := A0 + B0 + C0 + D0 + E0 + F0 + G0;

    constant INPUTS : PINSET := data;
    constant OUTPUTS : PINSET := OUT_PINS;
```

- Test vector filename declared in FILE statement.

The designer must modify the FILE statement in the waveform generator to include the name of the external test vector file (In this example, it was called "bcd_decoder_vectors.txt"). The FILE statement is in the format of VHDL '93. However, if needed, the FILE statement could be written in the VHDL '87 format. A variable called VECTOR is used to store a file slice read from the external file. A pinset data is declared for the 7-bit binary input signal. A pinset OUT_PINS is declared for all the decoder output signals. A pinset DISP1 is declared for the signals used to drive the upper digit of the display. A pinset DISP0 is declared for the signals used the drive the lower digit of the display. Finally, pinsets INPUTS and OUTPUTS are declared. [Hanna97]

Decoder Waveform Generator File (cont.)

```
variable TIMING : FRAME_DATA :=  
    BUILD_FRAME_DATA(  
        (  
            (INPUTS, NON_RETURN( 0 ns)),  
            (DISP1, WINDOW( 400 ns, 500 ns)),  
            (DISP0, WINDOW( 400 ns, 500 ns))  
        )  
    );  
  
begin -- waveform generator procedure  
  
    loop  
        READ_FILE_SLICE( VECTOR_FILE, VECTOR );  
        exit when VECTOR.END_OF_FILE;  
        APPLY( WPL, VECTOR.CODES.all, TIMING );  
        DELAY( VECTOR.FS_TIME );  
    end loop;  
  
end WAVEFORM;  
  
END WGP_display_driver;
```

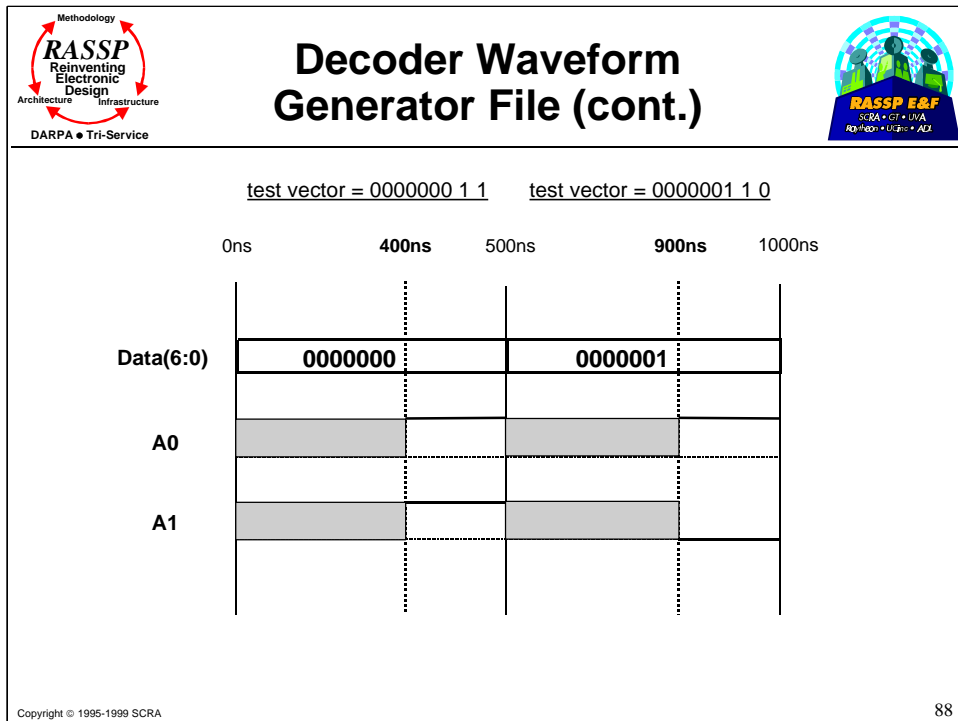
This slide shows the most important changes that the user must make to the waveform generator. In this section, the frame sets are created, and the test vectors are read and applied. The designer specifies how the frames are to be built using the various constructor functions (used in the function “Build_Frame_Data” above). The WAVES built-in functions are used to read and apply the various test vectors in the loop statement. This code will be examined in more detail in the next few slides.

Decoder Waveform Generator File (cont.)

- The user specifies the frame constructor functions and the appropriate timing values as shown below:

```
variable TIMING : FRAME_DATA :=  
    BUILD_FRAME_DATA(  
        (  
            (INPUTS, NON_RETURN( 0 ns)),  
            (DISP1, WINDOW( 400 ns, 500 ns)),  
            (DISP0, WINDOW( 400 ns, 500 ns))  
        )  
    );
```

The variable “TIMING” will store all the frame set information needed for the test pins on the decoder. “TIMING” is of type Frame_Data, which is a record of frame set arrays and test pin mapping information. The information for the variable “TIMING” is produced by the function Build_Frame_Data. The format Non_Return is used on the pinset INPUTS, which represents the 7 input pins. The Window format is used on the pinsets DISP1 and DISP0. A 100ns comparison interval has been declared. The constructor functions (Non_return and Window), along with their timing information, are used to create a data structure that is stored in the variable “TIMING”. The structure in “TIMING” will allow a frame to be selected and applied to a given test pin. The information in “TIMING” is indexed by a pin and pin code combination. [STD97]



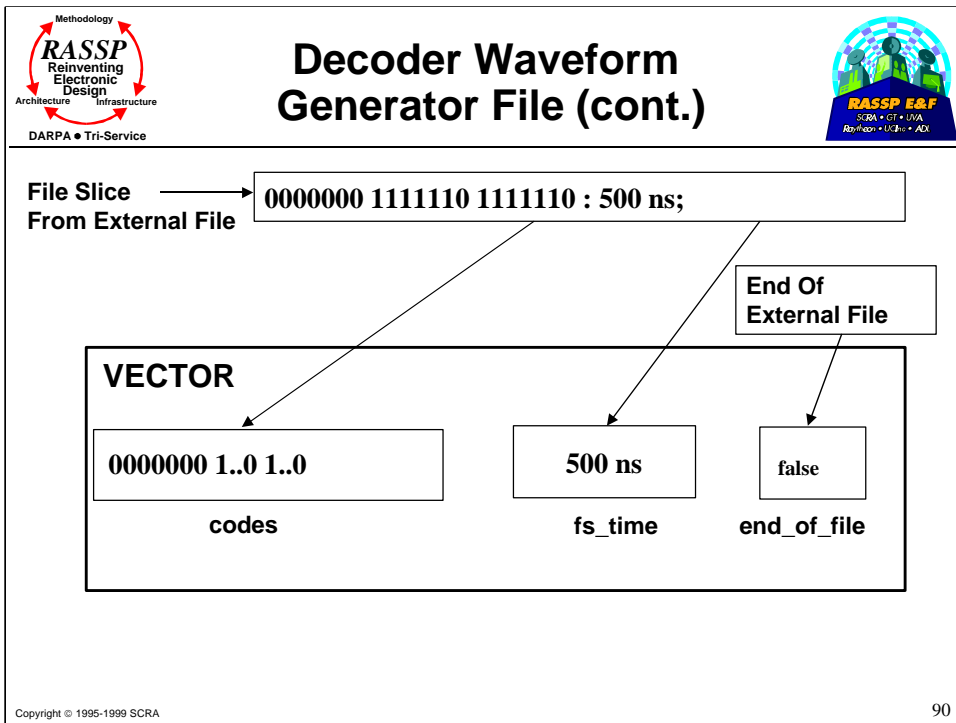
This diagram shows the impact of the constructor functions used in this waveform generator. The first two slices of the waveform are shown above. The test vectors shown are the excerpts from the first two test vectors from the external file. In reality, the entire test vector is 21 pin codes long. In the test vectors shown, the 7 pin codes for Data(6:0) are shown first. Then, the pin codes for A0 and A1 are shown. In the first frame (from 0ns to 500ns), the vector is “0000000 1 1”. The test vector causes the signal Data to take a value of “0000000” for the entire first slice as expected with the Non_return format. On the outputs A0 and A1, a 100ns comparison window is established. In the first slice, both expected output pin codes are ‘1’. This causes A0 and A1 to change to ‘1’ at 400ns. In the second frame (from 500ns to 1000ns), the vector is “0000001 1 0”. Since the pin codes for Data is “0000001”, the signal will remain at “0000001” for the entire slice. The pin code for A0 is ‘1’, but pin code for A1 is ‘0’. At 900 ns, A0 changes to ‘1’ while A1 changes to ‘0’. In summary, the constructor functions and timing values specified in the waveform generator create the format for each waveform that is applied to the test pins.

Decoder Waveform Generator File (cont.)



- In the loop, a slice is read from the external file and applied to the corresponding signals as shown below:

```
loop
    READ_FILE_SLICE( VECTOR_FILE, VECTOR );
    exit when VECTOR.END_OF_FILE;
    APPLY( WPL, VECTOR.CODES.all, TIMING );
    DELAY( VECTOR.FS_TIME );
end loop;
```

The LOOP statement shown above controls the reading and application of test vectors from the external file. The READ_FILE_SLICE procedure reads a file slice from the external file (specified in “vector_file”) and stores it in a variable “Vector”, which is actually a record of several fields. The information read from each file slice is stored in a specific field of “Vector”. An example of the various fields of “Vector” will be shown on the next slide. The APPLY procedure uses the pin code information from “Vector”, and the frame set structure in “TIMING” to schedule events on the test pins. The DELAY function uses the slice timing information contained in “Vector” to suspend execution of the WAVES test set. After this procedure is executed, the current time is effectively incremented by the value of the delay time. The loop is terminated when the Boolean end-of-file field in “Vector” is set. This flag will be set by READ_FILE_SLICE when it reaches the end of the external file. [STD97]



This diagram shows the basic fields that make up the record “Vector”. The information contained within each field is stored by the procedure `READ_FILE_SLICE`. The field “codes” is used to store the pin code information from a file slice. The “codes” field is used in the `APPLY` procedure shown on the previous slide. The slice timing information is stored in the “fs_time” field. The “fs_time” field is accessed by the `DELAY` procedure shown on the previous slide. If the procedure `READ_FILE_SLICE` encounters the end of the external file, then the “end_of_file” field is set in “Vector”. This field is essentially a Boolean flag that is set to True when the end of the external file has been reached. The “end_of_file” field is used by the `EXIT` statement seen on the previous slide. There are other fields in “Vector” that are not shown above. These fields would be used to support the Skip command, Repeat command, and hexadecimal formatting discussed in the External File section of this module. In summary, the information in a file slice is stored over several fields of “Vector”. Each field will be accessed by different WAVES functions sequence and construct the signals on the waveform. [STD97]



Decoder Testbench

```
-- ***** This File Was Automatically Generated *****
-- ***** By The WAVES96-VHDL Tool Set *****
-- ***** Generated for Entity: display_driver *****
-- ***** This File Was Generated on: Fri Jan 30 16:43:42 1998 *****
--
--

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.WAVES_1164_utilities.all;
USE IEEE.WAVES_interface.all;

USE WORK.UUT_test_pins.all;
USE work.waves_objects.all;
USE work.WGP_display_driver.all;

-- Include component library references here
-- User Must Modify And ADD component library references here
-- Include component library references here

ENTITY test_bench IS
END test_bench;
ARCHITECTURE display_driver_test OF test_bench IS
```

Copyright © 1995-1999 SCRA

91

The testbench can be automatically generated by the WAVES tools. However, the testbench **must be modified** before it can be simulated. The designer must insure that the library references for the testbench are correct. The designer must include the component library references. The testbench above uses the WAVES library, the test pin file, and the waveform generator file. An entity “test_bench” is declared with no ports in the entity declaration.

Decoder Testbench (cont.)

```
COMPONENT display_driver
  PORT ( data          : IN  std_logic_vector( 6 downto 0 );
        A0             : OUT std_logic;
        B0             : OUT std_logic;
        C0             : OUT std_logic;
        D0             : OUT std_logic;
        E0             : OUT std_logic;
        F0             : OUT std_logic;
        G0             : OUT std_logic;
        A1             : OUT std_logic;
        B1             : OUT std_logic;
        C1             : OUT std_logic;
        D1             : OUT std_logic;
        E1             : OUT std_logic;
        F1             : OUT std_logic;
        G1             : OUT std_logic);
  END COMPONENT;

-- User Must Modify modify and declare correct
-- .. Architecture, Library, Component ..
-- Modify entity use statement
FOR ALL:display_driver USE ENTITY work.display_driver(default);
```

The component under test, `display_driver`, is formally declared in the testbench. The user modifies the entity “use” statement to include the architecture name for the component (called “default” in this example).

Decoder Testbench (cont.)

```
--*****
-- stimulus signals for the waveforms mapped into UUT INPUTS
--*****
SIGNAL WAV_STIM_data          :std_logic_vector( 6 downto 0 );

--*****
-- Expected signals used in monitoring the UUT OUTPUTS
--*****
SIGNAL FAIL_SIGNAL            :std_logic;
SIGNAL WAV_EXPECT_A0          :std_ulogic;
SIGNAL WAV_EXPECT_B0          :std_ulogic;
SIGNAL WAV_EXPECT_C0          :std_ulogic;
SIGNAL WAV_EXPECT_D0          :std_ulogic;
SIGNAL WAV_EXPECT_E0          :std_ulogic;
SIGNAL WAV_EXPECT_F0          :std_ulogic;
SIGNAL WAV_EXPECT_G0          :std_ulogic;
SIGNAL WAV_EXPECT_A1          :std_ulogic;
SIGNAL WAV_EXPECT_B1          :std_ulogic;
SIGNAL WAV_EXPECT_C1          :std_ulogic;
SIGNAL WAV_EXPECT_D1          :std_ulogic;
SIGNAL WAV_EXPECT_E1          :std_ulogic;
SIGNAL WAV_EXPECT_F1          :std_ulogic;
SIGNAL WAV_EXPECT_G1          :std_ulogic;
```

Various signals are declared within the testbench to provide stimulus to the input signals and to monitor the output signals. The signal FAIL_SIGNAL is used to depict the status of the test in the monitor processes. In this example, there are 14 expected output signals declared for the decoder. [Hanna97]

Decoder Testbench (cont.)

```
-- *****
-- UUT Output signals used In Monitoring ACTUAL Values
-- *****
SIGNAL ACTUAL_A0      :std_logic;
SIGNAL ACTUAL_B0      :std_logic;
SIGNAL ACTUAL_C0      :std_logic;
SIGNAL ACTUAL_D0      :std_logic;
SIGNAL ACTUAL_E0      :std_logic;
SIGNAL ACTUAL_F0      :std_logic;
SIGNAL ACTUAL_G0      :std_logic;
SIGNAL ACTUAL_A1      :std_logic;
SIGNAL ACTUAL_B1      :std_logic;
SIGNAL ACTUAL_C1      :std_logic;
SIGNAL ACTUAL_D1      :std_logic;
SIGNAL ACTUAL_E1      :std_logic;
SIGNAL ACTUAL_F1      :std_logic;
SIGNAL ACTUAL_G1      :std_logic;

-- *****
-- WAVES signals OUTPUTing each slice of the waves port list
-- *****

SIGNAL wpl  : WAVES_port_list;
```

There are 14 signals declared for the actual output signals from the decoder.

Decoder Testbench (cont.)

```
BEGIN
--*****
-- process that generates the WAVES waveform
--*****
    WAVES: waveform(wpl);

--*****
-- processes that assign the WPL values to testbench signals
--*****
WAV_STIM_data      <= To_StdLogicVector(wpl.signals( 1 to 7 ));
WAV_EXPECT_A1      <= wpl.signals( 8 );
WAV_EXPECT_B1      <= wpl.signals( 9 );
WAV_EXPECT_C1      <= wpl.signals( 10 );
WAV_EXPECT_D1      <= wpl.signals( 11 );
WAV_EXPECT_E1      <= wpl.signals( 12 );
WAV_EXPECT_F1      <= wpl.signals( 13 );
WAV_EXPECT_G1      <= wpl.signals( 14 );
WAV_EXPECT_A0      <= wpl.signals( 15 );
WAV_EXPECT_B0      <= wpl.signals( 16 );
WAV_EXPECT_C0      <= wpl.signals( 17 );
WAV_EXPECT_D0      <= wpl.signals( 18 );
WAV_EXPECT_E0      <= wpl.signals( 19 );
WAV_EXPECT_F0      <= wpl.signals( 20 );
WAV_EXPECT_G0      <= wpl.signals( 21 );
```

The PROCEDURE defined in the waveform generator is used to generate a WAVES process in the testbench (in this example, the waveform generator procedure “waveform” is used). The waveform generator sends both the stimulus and expected response to the testbench through a Waves_Port_List signal type (called “wpl” in this example). The values from the Waves_Port_List must be translated into standard logic 1164 type. The assignment processes in this slide accomplish this task. [Hanna97]

Decoder Testbench (cont.)

```
-- *****  
-- UUT Port Map - Name Semantics Denote Usage  
-- *****  
ul: display_driver  
PORT MAP(  
  data          => WAV_STIM_data,  
  A0            => ACTUAL_A0,  
  B0            => ACTUAL_B0,  
  C0            => ACTUAL_C0,  
  D0            => ACTUAL_D0,  
  E0            => ACTUAL_E0,  
  F0            => ACTUAL_F0,  
  G0            => ACTUAL_G0,  
  A1            => ACTUAL_A1,  
  B1            => ACTUAL_B1,  
  C1            => ACTUAL_C1,  
  D1            => ACTUAL_D1,  
  E1            => ACTUAL_E1,  
  F1            => ACTUAL_F1,  
  G1            => ACTUAL_G1);
```

The display driver component is formally instantiated using structural VHDL. The testbench actual output signals are mapped to the outputs of the display driver.

Decoder Testbench (cont.)

```
--*****
-- Monitor Processes To Verify The UUT Operational Response
--*****
Monitor_A0:
PROCESS(ACTUAL_A0, WAV_expect_A0)
BEGIN
    assert(Compatible (actual => ACTUAL_A0,
                        expected => WAV_expect_A0))
    report "Error on A0 output" severity WARNING;
END PROCESS;

Monitor_B0:
PROCESS(ACTUAL_B0, WAV_expect_B0)
BEGIN
    assert(Compatible (actual => ACTUAL_B0,
                        expected => WAV_expect_B0))
    report "Error on B0 output" severity WARNING;
END PROCESS;
```

These processes monitor the value of the output signals by comparing the expected value from the waveform generator and the value produced by the VHDL component under test. There are 14 monitor processes in the testbench, one for each output signal on the UUT.

Decoder Testbench (cont.)

```
Monitor_C0:
PROCESS(ACTUAL_C0, WAV_expect_C0)
BEGIN
    assert(Compatible (actual => ACTUAL_C0,
                        expected => WAV_expect_C0))
    report "Error on C0 output" severity WARNING;
END PROCESS;

Monitor_D0:
PROCESS(ACTUAL_D0, WAV_expect_D0)
BEGIN
    assert(Compatible (actual => ACTUAL_D0,
                        expected => WAV_expect_D0))
    report "Error on D0 output" severity WARNING;
END PROCESS;

Monitor_E0:
PROCESS(ACTUAL_E0, WAV_expect_E0)
BEGIN
    assert(Compatible (actual => ACTUAL_E0,
                        expected => WAV_expect_E0))
    report "Error on E0 output" severity WARNING;
END PROCESS;
```

Decoder Testbench (cont.)

```
Monitor_F0:
PROCESS(ACTUAL_F0, WAV_expect_F0)
BEGIN
    assert(Compatible (actual => ACTUAL_F0,
                        expected => WAV_expect_F0))
    report "Error on F0 output" severity WARNING;
END PROCESS;

Monitor_G0:
PROCESS(ACTUAL_G0, WAV_expect_G0)
BEGIN
    assert(Compatible (actual => ACTUAL_G0,
                        expected => WAV_expect_G0))
    report "Error on G0 output" severity WARNING;
END PROCESS;

Monitor_A1:
PROCESS(ACTUAL_A1, WAV_expect_A1)
BEGIN
    assert(Compatible (actual => ACTUAL_A1,
                        expected => WAV_expect_A1))
    report "Error on A1 output" severity WARNING;
END PROCESS;
```

Decoder Testbench (cont.)

```
Monitor_B1:
PROCESS(ACTUAL_B1, WAV_expect_B1)
BEGIN
    assert(Compatible (actual => ACTUAL_B1,
                        expected => WAV_expect_B1))
    report "Error on B1 output" severity WARNING;
END PROCESS;

Monitor_C1:
PROCESS(ACTUAL_C1, WAV_expect_C1)
BEGIN
    assert(Compatible (actual => ACTUAL_C1,
                        expected => WAV_expect_C1))
    report "Error on C1 output" severity WARNING;
END PROCESS;

Monitor_D1:
PROCESS(ACTUAL_D1, WAV_expect_D1)
BEGIN
    assert(Compatible (actual => ACTUAL_D1,
                        expected => WAV_expect_D1))
    report "Error on D1 output" severity WARNING;
END PROCESS;
```

Decoder Testbench (cont.)

```
Monitor_E1:
PROCESS(ACTUAL_E1, WAV_expect_E1)
BEGIN
    assert(Compatible (actual => ACTUAL_E1,
                        expected => WAV_expect_E1))
    report "Error on E1 output" severity WARNING;
END PROCESS;

Monitor_F1:
PROCESS(ACTUAL_F1, WAV_expect_F1)
BEGIN
    assert(Compatible (actual => ACTUAL_F1,
                        expected => WAV_expect_F1))
    report "Error on F1 output" severity WARNING;
END PROCESS;


Monitor_G1:
PROCESS(ACTUAL_G1, WAV_expect_G1)
BEGIN
    assert(Compatible (actual => ACTUAL_G1,
                        expected => WAV_expect_G1))
    report "Error on G1 output" severity WARNING;
END PROCESS;
END display_driver_test;
```



Module Outline




- Introduction
- Testbench Development
- Design Verification Challenges
- WAVES Concepts
- WAVES Constructor Library and Built-Ins
- WAVES External File
- WAVES Test Set
- Decoder Example
- **Algorithmic Waveform Generator Example**



Methodology
RASSP
Reinventing
Electronic
Design
Infrastructure
DARPA • Tri-Service

Algorithmic Waveform Generators



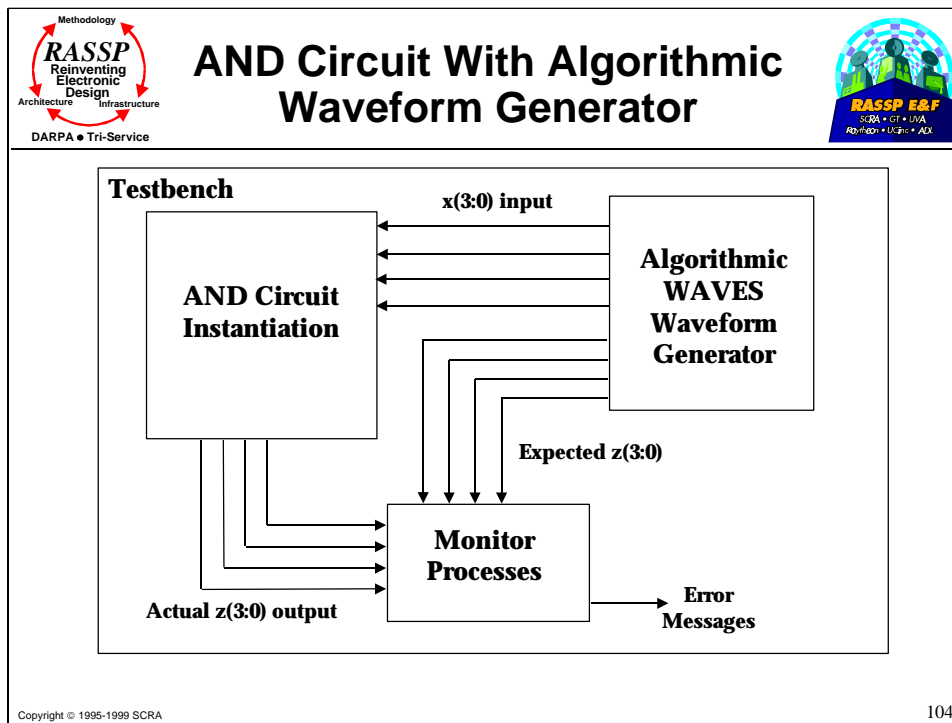
RASSP E&F
SCRA • CT • USA
Rapid • USF • AX

- **Test vectors can be created algorithmically in the waveform generator**
- **External file not required in this case**
- **All information that was previously contained in file slices is now generated in the waveform generator procedure**

Copyright © 1995-1999 SCRA

103

As stated earlier, the external pattern file is optional in the WAVES test set. If the pin codes can be generated algorithmically within the waveform generator, then the external file is not required. However, in many applications, it is difficult to write an algorithm in VHDL that could generate the appropriate test vectors. If such an algorithm can be written, then all the pin code and timing information that was previously contained within the test vectors must now be controlled within the waveform generator.



This example demonstrates an algorithmic waveform generator for a combinational AND circuit. The input stimulus and expected output values are generated entirely within the waveform generator. The algorithm for the test vectors is based on a Built-In Self Test (BIST) technique using Linear Feedback Shift Registers (LFSR). The LFSR will produce a pseudorandom sequence of bit values over several clock cycles. If designed properly, the LFSR will traverse all the possible logic states except for zero. In this example, the AND circuit has 4 inputs, which means the LFSR will generate a binary number between 1 and 15 every clock cycle. Once all 15 values have been produced, then the sequence of logic values is repeated. The use of LFSR allows certain portions of a hardware circuit to be tested using a known set of input vectors. The LFSR is a common method of including extra hardware components in order to support BIST. [Abramovici90]



AND Circuit With Algorithmic Waveform Generator Example




- **This WAVES Example Consists Of 5 Files:**
 - Header File
 - Behavioral VHDL Description Of AND Circuit
 - Test Pins File
 - Waveform Generator File
 - Testbench File
- **Algorithmic Waveform Generator Requires Additional Functionality Compared To Normal Waveform Generators**

Copyright © 1995-1999 SCRA

105


The algorithmic waveform generator example includes the 5 files shown above. There are several modifications required in an algorithmic waveform generator that are not required in normal waveform generators.



RASSP
Reinventing
Electronic
Design
Infrastructure

DARPA • Tri-Service

AND Circuit With Algorithmic Waveform Generator Header File



RASSP E&F
SCRA • GT • UVA
Rapid • U.S. • A&E

```
-- ***** Header File for Entity: and_ckt

-- Data Set Identification Information
TITLE      Algorithmic Waveform Generator Example
DEVICE_ID  and_ckt
DATE       Sun Sep 28 16:41:28 1997
ORIGIN     Advanced WAVES Module
AUTHOR     RASSP E&F and UVA
OTHER      This example demonstrates an algorithmic waveform
OTHER      generator which creates the test vectors without
OTHER      using an external file.

-- Data Set Construction Information
VHDL_FILENAME  a_ckt.vhd                                WORK
WAVES_FILENAME a_ckt_pins.vhd                            WORK
library
IEEE;
use            IEEE.WAVES_1164_Declarations.all;
use            IEEE.WAVES_Interface.all;
use            WORK.UUT_Test_pins.all;
WAVES_UNIT     WAVES_OBJECTS                             WORK
WAVES_FILENAME a_ckt_wgen.vhd                            WORK
VHDL_FILENAME  a_ckt_tstbench.vhd                        WORK
WAVEFORM_GENERATOR_PROCEDURE  WORK.wgp_and_ckt.waveform
```

Copyright © 1995-1999 SCRA

106

This is the header file for this example. The device under test is identified along with the various filenames in the WAVES test set. Note the absence of an external file.

AND Circuit VHDL Description

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY and_ckt IS
PORT (x: in std_ulogic_vector(3 downto 0);
      z: out std_ulogic_vector(3 downto 0));
END and_ckt;


ARCHITECTURE behave OF and_ckt IS
BEGIN
  PROCESS (x)
  BEGIN
    z(0) <= NOT(x(0)) AND NOT(x(2));
    z(1) <= NOT(x(2)) AND x(1);
    z(2) <= NOT(x(0)) AND NOT(x(3)) AND x(1);
    z(3) <= NOT(x(1)) AND x(2) AND x(3);
  END PROCESS;
END behave;
```

This is the VHDL description for AND circuit used in this example. Essentially, there are four AND gates in the circuit. Each AND gate produces one of the four output values for the circuit.

AND Circuit Test Pin File

```
-- ***** This File Was Automatically Generated *****  
-- ***** By The WAVES96-VHDL Tool Set *****  
-- ***** Generated for Entity: and_ckt *****  
-- ***** This File Was Generated on: Sun Sep 28 16:41:28 1997 *****  
--  
--  
PACKAGE uut_test_pins IS  
  TYPE test_pins IS (x_3, x_2, x_1, x_0, z_3, z_2, z_1, z_0);  
END uut_test_pins;
```


This is the test pin file for this example. There are eight test pins in the design: four input pins and four output pins.



RASSP
Reinventing
Electronic
Design
Architecture Infrastructure

DARPA • Tri-Service

AND Circuit Algorithmic Waveform Generator



RASSP E&F
SCRA • GT • USA
Rapid • U.S. • ALX

```
-- ***** This File Was Automatically Generated *****
-- ***** By The WAVES96-VHDL Tool Set *****
-- ***** Generated for VHDL entity: *****
-- ***** and_ckt *****
-- ***** Generation date and time: *****
-- ***** Sun Sep 28 16:41:28 1997 *****

use STD.TEXTIO.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_1164_extensions.all;
use IEEE.WAVES_1164_Frames.all;
use IEEE.WAVES_1164_Declarations.all;
use IEEE.WAVES_Interface.all;
use WORK.WAVES_Objects.all;
use WORK.UUT_Test_Pins.all;

package WGP_and_ckt is
    procedure WAVEFORM( signal WPL : inout WAVES_PORT_LIST );
end WGP_and_ckt;
```

Copyright © 1995-1999 SCRA
109

This is the algorithmic waveform generator for the AND circuit. In the package definition statement, only the procedure Waveform appears. The procedure Waveform is the only member of package WGP_and_ckt that will be called by the testbench. Note that the 1164 logic extensions package has been included in this waveform generator.

AND Circuit Algorithmic Waveform Generator


```
package body WGP_and_ckt is

    FUNCTION lfsr (CONSTANT x : IN waves_logic_vector(3 DOWNT0 0))
        RETURN waves_logic_vector IS
    VARIABLE z: waves_logic_vector(3 DOWNT0 0);
    BEGIN
        z(3 downto 1) := x(2 downto 0);
        z(0) := x(3) xor x(0);
        RETURN z;
    END lfsr;

    procedure WAVEFORM( signal WPL : inout WAVES_PORT_LIST ) is
    constant x : PINSET := x_3 + x_2 + x_1 + x_0;
    constant z : PINSET := z_3 + z_2 + z_1 + z_0;


    variable lfsr_array: waves_logic_vector
        ((test_pins'pos(test_pins'left) + 1)
         to (test_pins'pos(test_pins'right) + 1));
```

In order to algorithmically produce the pin codes for the AND circuit, several changes were made to the WGP_and_ckt package. The function lfsr shown above implements the LFSR algorithm described earlier. Basically, this function produces the next set of input values based on the current input values. Waves_logic_vector is defined as type in the WAVES 1164 packages. It is equivalent to std_ulogic_vector. This slide also shows the beginning of procedure Waveform. Two pinsets are declared for the input pins and output pins, respectively. A variable lfsr_array is declared of type waves_logic_vector. The length of this variable is defined by the number of test pins on the UUT. In this example, lfsr_array contain 8 logic values. Lfsr_array will be used in the waveform generator to store the logic values for each test pin. [STD97]



RASSP
Reinventing
Electronic
Design
Architecture Infrastructure
DARPA • Tri-Service

AND Circuit Algorithmic Waveform Generator



RASSP E&F
SCRA • GT • USA
Reinventing • U.S. • ALX

```

type lfsr_values is array(1 to 15) of
                                waves_logic_vector(3 downto 0);
constant z_array : lfsr_values :=
  ( ("0000"), ("0111"), ("0010"), ("0000"), ("0000"), ("0100"),
    ("0000"), ("0001"), ("0000"), ("0011"), ("0010"), ("1000"),
    ("1000"), ("0000"), ("0000"));
variable i : positive := 1;
variable j : integer := 0;
variable x_count : integer := 0;

variable pincode_values : pin_code_string;
constant period : delay_time := 20 ns;
variable TIMING : FRAME_DATA :=
  BUILD_FRAME_DATA(
    (
      (x, NON_RETURN(0 ns)),
      (z, WINDOW(10 ns, 15 ns))
    )
  );

```

Copyright © 1995-1999 SCRA

111

An array type called `lfsr_values` is declared, followed by a constant array called `z_array`. This constant array is used to store the expected output values. The expected output values for the AND circuit were generated through simulation of the design. The values in `z_array` are ordered by their corresponding input value. For example, for the input pattern "0010", the corresponding output value "0111" is stored in array position 2. This allows the expected output values to be retrieved based on the current input values. Two integer variables are declared next. These variables are used as loop indexes later in the procedure. The variable `x_count` is used to store the number of test pins associated with the pinset `x`. The variable `pincode_values` is declared of type `pin_code_string`. The length of `pincode_values` is determined by the number of test pins on the UUT. In this example, `pincode_values` will be a string of 8 characters. The constant period is used to define the slice timing information for this test set. Finally, the frame format definition is shown for pinsets `x` and `z`. The Non Return format appears on `x`, while the Window format appears on `z`. [STD97]


AND Circuit Algorithmic Waveform Generator

```
begin -- waveform generator procedure

    lfsr_array := "01100000"; -- initialize lfsr state
    for j in x'range loop
        if (x(test_pins(j)) = TRUE) then
            x_count := x_count + 1;
        end if;
    end loop;

    loop
        lfsr_array((test_pins'pos(test_pins'left)+ 1) to x_count):=
            lfsr(lfsr_array((test_pins'pos(test_pins'left) + 1) to
                x_count));
        lfsr_array(((test_pins'pos(test_pins'left) + 1) + x_count)
            to (test_pins'pos(test_pins'right) + 1)) :=
            z_array(To_Integer('0' &
                lfsr_array((test_pins'pos(test_pins'left) + 1) to
                    x_count)));
    end loop;
```


This slide shows the initialization of the waveform generator. First, `lfsr_array` is set to an initial starting point for the LFSR algorithm. Since the LFSR algorithm does not produce “0000” at any point, `lfsr_array` must be initialized to a non-zero value. In this example, the first four elements of `lfsr_array` are initialized “0110”. The first four elements correspond to the logic values for the four input pins on the UUT. Next, a loop statement is used to set `x_count`. In order to set the proper limits on the array variables later on, it is necessary to find the number of input pins in pinset `x`. The control loop for procedure `Waveform` is shown next. First, the next set of input values is computed using function `lfsr`. The current input values, stored in the first four positions of `lfsr_array`, are passed to function `lfsr`. The new input values are then stored in the first four positions of `lfsr_array`. Note how `x_count` is used to set the upper bound on the array values. The next operation is to convert the four input values into an integer. This is done using the `To_Integer` function found in the 1164 extensions package. An extra ‘0’ is added to the logic values when calling `To_Integer` in order to assure a positive result. Next, the integer is used to retrieve the expected output values from `z_array`. These values are then stored in the last four elements of `lfsr_array`. Note how `x_count` is used to set the limits on the array elements. After this step, `lfsr_array` contains all the logic values for each UUT test pin.



RASSP
Reinventing
Electronic
Design
Infrastructure

DARPA • Tri-Service

AND Circuit Algorithmic Waveform Generator



RASSP E&F
SCRA • GT • USA
Reinventing • U.S. • ALX

```

for i in (test_pins'pos(test_pins'left) + 1) to
    (test_pins'pos(test_pins'right) + 1) loop
    pincode_values(i) :=
        pin_codes(logic_value'pos(lfsr_array(i)));
    end loop;
    APPLY( WPL, pincode_values, TIMING );
    DELAY( period );
end loop;
end WAVEFORM;
END WGP_and_ckt;

```

Copyright © 1995-1999 SCRA

113

The next step is to convert the logic values in `lfsr_array` into a string of characters. Normally, this would be done by the function `Read_File_Slice` when it reads test vectors from a file. However, an explicit conversion is necessary in this example. Each logic value in `lfsr_array` is converted to its equivalent string character in the loop statement. The characters are defined as the legal pin code values in the 1164 Waves packages. The string of pin code characters is stored in `pincode_values`. Next, procedure `Apply` is called. Procedure `Apply` schedules the appropriate events on the test pins. Finally, the `Delay` procedure uses the constant period to define the slice duration of 20ns. Each slice is defined by one pass through the main loop statement in procedure `Waveform`. This waveform generator was written in a generic way to support changes in the design. For example, if the number of test pins changed, only the expected output value array and initialization of `lfsr_array` would have to be modified. The main control loop would not require any changes since the array bounds are determined by the number of test pins. In summary, an algorithmic waveform generator will probably require additional functions and conversions in order to properly set up the pin codes for each slice. In this case, the use of an LFSR algorithm greatly simplified the production of inputs and expected outputs for the UUT. [STD97]

AND Circuit Testbench

```
-- ***** This File Was Automatically Generated *****
-- ***** By The WAVES96-VHDL Tool Set *****
-- ***** Generated for Entity: and_ckt *****
-- ***** This File Was Generated on: Sun Sep 28 16:41:29 1997 *****

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.WAVES_1164_utilities.all;
USE IEEE.WAVES_interface.all;
USE WORK.UUT_test_pins.all;
USE work.waves_objects.all;
USE work.WGP_and_ckt.all;
-- Include component library references here
-- User Must Modify And ADD component library references here
-- Include component library references here

ENTITY test_bench IS
END test_bench;

ARCHITECTURE and_ckt_test OF test_bench IS
```

The next several slides show the testbench for the AND circuit. In this example, no extra modifications were required in the testbench to support the algorithmic waveform generator.

AND Circuit Testbench (cont.)

```
-- *****
-- *****CONFIGURATION SPECIFICATION *****
-- *****

COMPONENT and_ckt
  PORT ( x                : IN  std_ulogic_vector( 3 downto 0 );
        z                : OUT  std_ulogic_vector( 3 downto 0 ));
  END COMPONENT;

-- Modify entity use statement
-- User Must Modify modify and declare correct
-- .. Architecture, Library, Component ..
-- Modify entity use statement
FOR ALL:and_ckt USE ENTITY work.and_ckt(behave);

-- *****
-- stimulus signals for the waveforms mapped into UUT INPUTS
-- *****

SIGNAL WAV_STIM_x          :std_ulogic_vector( 3 downto 0 );
```

The component `and_ckt` is formally declared in the Component statement seen above. The signal `Wav_Stim_x` is declared to stimulate the input pins on the UUT.

AND Circuit Testbench (cont.)

```
-- *****
-- Expected signals used in monitoring the UUT OUTPUTS
-- *****

SIGNAL FAIL_SIGNAL          :std_logic;
SIGNAL WAV_EXPECT_z         :std_ulogic_vector( 3 downto 0 );

-- *****
-- UUT Output signals used In Monitoring ACTUAL Values
-- *****

SIGNAL ACTUAL_z             :std_ulogic_vector( 3 downto 0 );

-- *****
-- WAVES signals OUTPUTing each slice of the waves port list
-- *****

SIGNAL wpl  : WAVES_port_list;
```

The signal Wav_Expect_z and Actual_z are declared to capture the expected and actual values on the UUT output.

AND Circuit Testbench (cont.)

```
BEGIN
--*****
-- process that generates the WAVES waveform
--*****
    WAVES: waveform(wpl);

--*****
-- processes that assign the WPL values to testbench signals
--*****
WAV_STIM_x      <= wpl.signals( 1 to 4 );
WAV_EXPECT_z    <= wpl.signals( 5 to 8 );

--*****
-- UUT Port Map - Name Semantics Denote Usage
--*****
u1: and_ckt
PORT MAP(
    x              => WAV_STIM_x,
    z              => ACTUAL_z);
```

The procedure Waveform is formally called in the testbench. This procedure call will trigger the various LFSR functions within the previously shown waveform generator package when Waveform is executing.

AND Circuit Testbench (cont.)

```
--*****
-- Monitor Processes To Verify The UUT Operational Response
--*****

Monitor_z:
  PROCESS(ACTUAL_z, WAV_expect_z)
  BEGIN
    assert(Compatible (actual => ACTUAL_z,
                      expected => WAV_expect_z))
    report "Error on z output" severity WARNING;

    IF ( Compatible ( ACTUAL_z,    WAV_expect_z ) ) THEN
      FAIL_SIGNAL <='L'; ELSE FAIL_SIGNAL <='1';
    END IF;
  END PROCESS;

END and_ckt_test;
```

This monitor process compares the expected and actual values on the output bus z.



References



- [Abramovici90] Abramovici, Miron, Melvin A. Breuer, Arthur D. Friedman. Digital Systems Testing And Testable Design. Computer Science Press, New York, 1990.
- [Flynn94] Christopher J. Flynn, Frederick G. Hall, James P. Hanna, and Mark T. Pronobis. "Using WAVES In A Top-Down Design Methodology," VHDL International Users' Forum, Nov. 1994.
- [Hanna97] Hanna, James P., Robert G. Hillman, Herb L. Hirsch, Tim H. Noh, Ranga R. Vemuri. Using WAVES And VHDL For Effective Design And Testing. Kluwer Academic Publishers, Boston, 1997.
- [IEEE] All referenced IEEE material is used with permission.
- [Pronobis95] Mark T. Pronobis, Robert Hillman, Christopher Flynn. "Test Insertion Without Being A Test Expert," VHDL International Users' Forum, Oct. 1995.
- [STD97] Draft IEEE Standard For VHDL Waveform And Vector Exchange (WAVES), IEEE Standard 1029.1-1996, IEEE Computer Society & IEEE Standards Coordinating Committee 20, May 1997.