9. Writing Test Benches

As you become proficient with simulation, you will find that your VHDL simulator becomes your primary design tool.

While much of this book has focused on the uses of VHDL for synthesis, one of the primary reasons to use VHDL is its power as a test stimulus language. As logic designs become more complex, comprehensive, up-front verification becomes critical to the success of a design project. In fact, as you become proficient with simulation, you will quickly find that your VHDL simulator becomes your primary design development tool. When simulation is used right at the start of the project, you will have a much easier time with synthesis, and you will spend far less time re-running time-intensive processes, such as FPGA place-and-route tools and other synthesis-related software.

To simulate your project, you will need to develop an additional VHDL program called a test bench. (Some VHDL simulators include a command line stimulus language, but these features are no replacement for a true test bench.) Test benches emulate a hardware breadboard into which you will "install" your synthesizable design description for the purpose of verification. Test benches can be quite simple, applying a sequence of inputs to the circuit over time. They can also be quite complex, perhaps even reading test data from a disk file

and writing test results to the screen and to a report file. A comprehensive test bench can, in fact, be more complex and lengthy (and take longer to develop) than the synthesizable circuit being tested. As you will begin to appreciate while reading this chapter, test bench development will be where you make use of the full power of VHDL and your own skills as a VHDL "coder".

Depending on your needs (and whether timing information related to your target device technology is available), you may develop one or more test benches to verify the design functionally (with no delays), to check your assumptions about timing relationships (using estimates or unit delays), or to simulate with annotated post-route timing information so you can verify that your circuit will operate in-system at speed.

During simulation, the test bench will be the top level of a design hierarchy. To the simulator, there is no distinction between those parts of the design that are being tested and the test bench itself. In your own mind, however, you can think of the test bench as a separate circuit, analogous to a large automated tester (Figure 9-1).

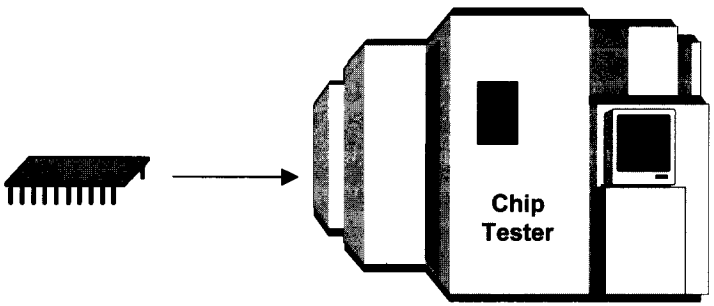

***Figure 9-1**: A test bench can be thought of as a "virtual tester" into which you plug your design for verification.*

In most of this book, we have been emphasizing those aspects of the VHDL language that are synthesizable. In doing so, we have actually seen only a subset of the VHDL language in the examples presented. When writing test benches, you will most likely use a broader range of language features. You may use records and multi-dimensional arrays to describe test

stimuli, write loops, create subprograms to simplify repetitive actions, and/or use VHDL's text I/O features to read and write files of data.

A Simple Test Bench

The simplest test benches apply some sequence of inputs to the circuit...

The simplest test benches are those that apply some sequence of inputs to the circuit being tested (the *Unit Under Test*, or *UUT*) so that its operation can be observed in simulation. Waveforms are typically used to represent the values of signals in the design at various points in time. Such a test bench must consist of a component declaration corresponding to the unit under test, and a description of the input stimulus being applied to the UUT.

The following example demonstrates the simplest form of a test bench, and tests the operation of a NAND gate:

```
library ieee;                    -- Load the ieee 1164 library
use ieee.std_logic_1164.all;     -- Make the package 'visible'

use work.nandgate;               -- We'll use the NAND gate model from 'work'

-- The top level entity of the test bench has no ports...
--
entity testnand is
end testnand;

architecture stimulus of testnand is
  -- First, declare the lower-level entity...
  component nand
    port  (A,B: in std_logic;
           Y: out std_logic);
  end component;

  -- Next, declare some local signals to assign values to and observe...
  signal A,B: std_logic;
  signal Y: std_logic;

begin
  -- Create an instance of the comparator circuit...
  NAND1: nandgate port map(A => A,B => B,Y => Y);
```

```
  -- Now define a process to apply some stimulus over time...
  process
    constant PERIOD: time := 40 ns;
  begin
    A <= '1';
    B <= '1';
    wait for PERIOD;
    assert (Y = '0')
      report "Test failed!" severity ERROR;
    A <= '1';
    B <= '0';
    wait for PERIOD;
      assert (Y = '1')
      report "Test failed!" severity ERROR;
    A <= '0';
    B <= '1';
    wait for PERIOD;
      assert (Y = '1')
      report "Test failed!" severity ERROR;
    A <= '0';
    B <= '0';
    wait for PERIOD;
      assert (Y = '1')
      report "Test failed!" severity ERROR;
    wait;
  end process;
end stimulus;
```

Reading from the top of this test bench, we see:

- **Library** and **use** statements making the standard logic package available for use (our lower-level NAND gate model has been described using standard logic).
- An optional **use** statement referencing the lower-level design unit **nand** from the **work** library.
- An entity declaration for the test bench. Note that test benches do not generally include an interface (port) list, as they are the highest-level design unit when simulated.
- An architecture declaration, containing:

- A component declaration corresponding to the unit under test.
- Signal declarations for **A**, **B**, and **Y**. These local signals will be used to (1) apply inputs to the unit under test, and (2) observe the behavior or the output during simulation.
- A component instantiation statement and corresponding port map statement that associates the top-level signals **A**, **B** and **Y** with their equivalent ports in the lower-level entity. Note that the component name used (**UUT**) is not significant; any valid component name could have been chosen.
- A **process** statement describing the inputs to the circuit over time. This process has been written without the use of a sensitivity list. It uses **wait** statements to provide a specific amount of delay (defined using constant **PERIOD**) between each new combination of inputs. **Assert** statements are used to verify that the circuit is operating correctly for each combination of inputs. Finally, a **wait** statement without any condition expression is used to suspend simulation indefinitely after the desired inputs have been applied. (In the absence of the final **wait** statement, the process would repeat forever, or for as long as the simulator had been instructed to run.)

Using Assert Statements

***Assert** statements provide a quick and easy way to check expected values...*

VHDL's **assert** statement provides a quick and easy way to check expected values and display messages from your test bench. An **assert** statement has the following general format:

assert *condition_expression*
  report *text_string*
  severity *severity_level*;

When analyzed (either during execution as a sequential statement, or during simulator initialization in the case of a concurrent **assert** statement), the condition expression is evaluated. As in an **if** statement, the condition expression of an **assert** statement must evaluate to a boolean (**true** or **false**) value. If the condition expression is **false** (indicating the assertion *failed*), the text that you have specified in the optional **report** statement clause is displayed in your simulator's transcript (or other) window. The **severity** statement clause then indicates to the simulator what action (if any) should be taken in response to the assertion failure (or *assertion violation*, to use the language of the VHDL specification).

The severity level can be specified using one of the following predefined severity levels: **NOTE**, **WARNING**, **ERROR**, or **FAILURE**. The actions that result from the use of these severity levels will depend on the simulator you are using, but you can generally expect the simulator to display a file name and line number associated with the **assert** statement, keep track of the number of assertion failures, and print a summary at the end of the simulation run. **Assert** statements that specify **FAILURE** in their severity statement clauses will normally result in the simulator halting.

Displaying Complex Strings in Assert Statements

VHDL's built-in support for formatted strings is somewhat limited...

A common use of **assert** and **report** statements is to display information about signals or variables dynamically during a simulation run. Unfortunately, VHDL's built-in support for this is somewhat limited. The problem is twofold: first, the **report** clause only accepts a single string as its argument, so it is necessary to either write multiple **assert** statements to output multiple lines of information (as when formatting and displaying a table), or you must make use of the string concatenation operator **&** and the special character constant **CR** (carriage return) and/or **LF** (line feed) to describe a single, multi-line string as shown below:

```
assert false
  report "This is the first line of the message." & CR & LF &
         "This is the second line of the message.";
```

The second, more serious limitation of the **report** statement clause is that it only accepts a string, and there is no built-in provision for formatting various types of data (such as arrays, integers and the like) for display. This means that to display such data in an **assert** statement, you must provide type conversion functions that will convert from the data types you are using to a formatted string. The following example (which is described in more detail later in this chapter) demonstrates how you might write a conversion function to display a **std_logic_vector** array value as a string of characters:

A conversion function can be used to display an array as a formatted string.

```
architecture stimulus of testfib is
  . . .
  function vec2str(vec: std_logic_vector) return string is
  variable stmp: string(vec'left+1 downto 1);
  begin
    for i in vec'reverse_range loop
      if (vec(i) = 'U') then
        stmp(i+1) := 'U';
      elsif (vec(i) = 'X') then
        stmp(i+1) := 'X';
      elsif (vec(i) = '0') then
        stmp(i+1) := '0';
      elsif (vec(i) = '1') then
        stmp(i+1) := '1';
      elsif (vec(i) = 'Z') then
        stmp(i+1) := 'Z';
      elsif (vec(i) = 'W') then
        stmp(i+1) := 'W';
      elsif (vec(i) = 'L') then
        stmp(i+1) := 'L';
      elsif (vec(i) = 'H') then
        stmp(i+1) := 'H';
      else
        stmp(i+1) := '-';
      end if;
    end loop;
  return stmp;
  end;
  . . .
```

```
  signal S: std_logic_vector(15 downto 0);
  signal S_expected: std_logic_vector(15 downto 0);
begin
  ...
  process
  begin
    ...

    assert (S /= S_expected)   -- report an error if different
          report "Vector failure!" & CR & LF &
          "Expected S to be  " & vec2str(S_expected) & CR & LF &
          "but its value was " & vec2str(S)
          severity ERROR;
```

In this example, a type conversion function has been written (**vec2str**) that converts an object of type **std_logic_vector** to a string of the appropriate format and size for display. As you develop more advanced test benches, you will probably find it useful to collect such type conversion functions into a library for use in future test benches.

As we will see later in this chapter, there are other, more powerful ways to display formatted output, using the built-in text I/O features of the language.

Using Loops and Multiple Processes

Loops can dramatically simplify a test bench.

Test benches can be dramatically simplified through the use of loops, constants and other more advanced features of VHDL. Using multiple concurrent processes in combination with loops can result in very concise descriptions of complex input and expected output conditions.

The following example demonstrates how a loop (in this case a **while** loop) might be used to create a background clock in one process, while other loops (in this case **for** loops) are used to apply inputs and monitor outputs over potentially long periods of time:

```
Clock1: process
  variable clktmp: std_logic := '1';
begin
```

```
    while done /= true loop
      wait for PERIOD/2;
      clktmp := not clktmp;
      Clk <= clktmp;
    end loop;
    wait;
  end process;

  Stimulus1: Process
  Begin
    Reset <= '1';
    wait for PERIOD;
    Reset <= '0';
    Mode <= '0';
    wait for PERIOD;
    Data <= (others => '1');
    wait for PERIOD;
    Mode <= '1';

    -- Check to make sure we detect the vertical sync...
    Data <= (others => '0');
    for i in 0 to 127 loop
      wait for PERIOD;
      assert (VS = '1')
        report "VS went high at the wrong place!" severity ERROR;
    end loop;
    assert (VS = '1')
      report "VS was not detected!" severity ERROR;

    -- Load in the test counter value to check the end of frame detection...
    TestLoad <= '1';
    wait for PERIOD;
    TestLoad <= '0';
    for i in 0 to 300 loop
      Data <= RandomData();
      wait for PERIOD;
    end loop;
    assert (EOF = '1')
      report "EOF was not detected!" severity ERROR;

    done <= true;
    wait;

  End Process;

End stimulus;
```

In this example, the process labeled **Clock1** uses a local variable (**clktmp**) to describe a repeating clock with a period defined by the constant **PERIOD**. This clock is described with a **while** loop statement, and it runs independent of all other processes in the test bench until the **done** signal is asserted **true**. The second process, **Stimulus1**, describes a sequence of inputs to be applied to the unit under test. It also makes use of loops—in this case **for** loops—to describe lengthy repeating stimuli and expected value checks.

Writing Test Vectors

Test vectors are sequences of inputs and corresponding outputs expressed in tabular form.

Another approach to creating test stimuli is to describe the test bench in terms of a sequence of fixed input and expected output values. This sequence of values (sometimes called *test vectors*) could be described using multi-dimensional arrays or using arrays of records. The following example makes use of a record data type, **test_record**, which consists of the record elements **CE**, **Set**, **Din** and **CRC_Sum**. An array type (**test_array**) is then declared, representing an unconstrained array of **test_record** type objects. The constant **test_vectors**, of type **test_array**, is declared and assigned values corresponding to the inputs and expected output for each desired test vector.

The test bench operation is described using a **for** loop within a process. This **for** loop applies the input values **Set** and **Din** (from the test record corresponding to the current iteration of the loop) to the unit under test. (The **CE** input is used within the test bench to enable or disable the clock, and is not passed into the unit under test.) After a certain amount of time has elapsed (as indicated by a **wait** statement), the **CRC_Sum** record element is compared against the corresponding output of the unit under test, using an **assert** statement.

```
library ieee;
use ieee.std_logic_1164.all;

use work.crc8s;    -- Get the design out of library 'work'
```

```
entity testcrc is
end testcrc;

architecture stimulus of testcrc is
  component crc8s
    port (Clk,Set,Din: in std_logic;
          CRC_Sum: out std_logic_vector(15 downto 0));
  end component;

  signal CE: std_logic;
  signal Clk,Set: std_logic;
  signal Din: std_logic;
  signal CRC_Sum: std_logic_vector(15 downto 0);
  signal vector_cnt: integer := 1;
  signal error_flag: std_logic := '0';

  type test_record is record                    -- Declare a record type
    CE: std_logic;                              -- Clock enable
    Set: std_logic;                             -- Register preset signal
    Din: std_logic;                             -- Serial Data input
    CRC_Sum: std_logic_vector (15 downto 0);    -- Expected result
  end record;

  type test_array is array(positive range <>) of test_record;  -- Collect them
                                                               -- in an array

  -- The following constant declaration describes the test vectors to be
  -- applied to the design during simulation, and the expected result after a
  -- rising clock edge.
  constant test_vectors : test_array := (
    -- CE, Set, Din, CRC_Sum
      ('0', '1', '0', "----------------"),  -- Reset

      ('1', '0', '0', "----------------"), -- 'H'
      ('1', '0', '1', "----------------"),
      ('1', '0', '0', "----------------"),
      ('1', '0', '0', "----------------"),
      ('1', '0', '1', "----------------"),
      ('1', '0', '0', "----------------"),
      ('1', '0', '0', "----------------"),
      ('1', '0', '0', "0010100000111100"), -- x283C

      ('1', '0', '0', "----------------"), -- 'e'
      ('1', '0', '1', "----------------"),
      ('1', '0', '1', "----------------"),
      ('1', '0', '0', "----------------"),
```

Record data types can be useful for representing test vector data.

```
        ('1', '0', '0', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '0', "----------------"),
        ('1', '0', '1', "1010010101101001"), -- xA569

        ('1', '0', '0', "----------------"), -- 'l'
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '0', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '0', "----------------"),
        ('1', '0', '0', "0010000101100101"), -- x2165

        ('1', '0', '0', "----------------"), -- 'l'
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '0', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '0', "----------------"),
        ('1', '0', '0', "1111110001101001"), -- xFC69

        ('1', '0', '0', "----------------"), -- 'o'
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '0', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "----------------"),
        ('1', '0', '1', "1101101011011010")  -- xDADA
    );

begin
    -- instantiate the component
    UUT: crc8s port map(Clk,Set,Din,CRC_Sum);

    -- provide stimulus and check the result

    testrun: process
        variable vector : test_record;
    begin
        for index in test_vectors'range loop
            vector_cnt <= index;
            vector := test_vectors(index);    -- Get the current test vector
                    -- Apply the input stimulus...
```

```
        CE <= vector.CE;
        Set <= vector.Set;
        Din <= vector.Din;

            -- Clock (low-high-low) with a 100 ns cycle...
        Clk <= '0';
        wait for 25 ns;
        if CE = '1' then
          Clk <= '1';
        end if;
        wait for 50 ns;
        Clk <= '0';
        wait for 25 ns;

            -- Check the results...
        if (vector.CRC_Sum /= "----------------"
            and CRC_Sum  /= vector.CRC_Sum) then
          error_flag <= '1';
          assert false
            report "Output did not match!"
            severity WARNING;
        else
          error_flag <= '0';
        end if;
      end loop;
      wait;
    end process;
end stimulus;
```

Note:

VHDL 1076-1993 broadens the scope of bit string literals somewhat, making it possible to enter ***std_logic_vector*** *data in non-binary forms, as in the constant hexadecimal value* ***x"283C"****.*

Reading and Writing Files with Text I/O

Text I/O features allow you to read and write files in your test bench.

The text I/O features of VHDL make it possible to open one or more data files, read lines from those files, and parse the lines to form individual data elements, such as elements in an array or record. To support the use of files, VHDL has the concept of a **file** data type, and includes standard, built-in functions for opening, reading from, and writing to file data types. (These

data types and functions were described in Chapter 3, *Exploring Objects and Data Types*.) The **textio** package, which is included in the standard library, expands on the built-in file type features by adding text parsing and formatting functions, functions and special file types for use with interactive ("std_input" and "std_output") I/O operations, and other extensions.

Text I/O is one area in which the 1076-1987 and 1076-1993 language specifications differ. The file and text I/O features of VHDL changed in the 1076-1993 standard, making it necessary to explicitly open a file before reading data from it. We will show how file I/O can be described using both the 1987 and 1993 language standards.

VHDL 1076-1987 File I/O

The following example demonstrates how you can use the text I/O features of VHDL to read test data from an ASCII file, using the VHDL 1076-1987 standard text I/O features.

This test bench reads lines from an ASCII file and applies the data contained in each line as a test vector to stimulate and test a simple Fibonacci sequence generator circuit. It begins with our by-now-familiar entity-architecture pair:

*To use the text I/O features of the standard library, you must include a **use** reference to package **std.textio**.*

```
------------------------------------------------------------
-- Test bench for Fibonacci sequence generator.
--
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;          -- Use the text I/O features of the standard library
use work.fib;                -- Get the design out of library 'work'
entity testfib is            -- Entity; once again we have no ports
end testfib;

architecture stimulus of testfib is
  component fib              -- Create one instance of the fib design unit
    port (Clk,Clr: in std_logic;
          Load: in std_logic;
          Data_in: in std_logic_vector(15 downto 0);
          S: out std_logic_vector(15 downto 0));
  end component;
```

```
-- Define some local conversion functions. These will be used to:
--   (a) convert strings of characters read from the file into arrays
--       of values
--   (b) convert arrays of std_logic to strings for display purposes
--
function str2vec(str: string) return std_logic_vector is
variable vtmp: std_logic_vector(str'range);
begin
  for i in str'range loop
    if (str(i) = '1') then
      vtmp(i) := '1';
    elsif (str(i) = '0') then
      vtmp(i) := '0';
    else
      vtmp(i) := 'X';
    end if;
  end loop;
  return vtmp;
end;

function vec2str(vec: std_logic_vector) return string is
variable stmp: string(vec'left+1 downto 1);
begin
  for i in vec'reverse_range loop
    if (vec(i) = '1') then
      stmp(i+1) := '1';
    elsif (vec(i) = '0') then
      stmp(i+1) := '0';
    else
      stmp(i+1) := 'X';
    end if;
  end loop;
return stmp;
end;

signal Clk,Clr: std_logic;          -- Declares local signals
signal Load: std_logic;
signal Data_in: std_logic_vector(15 downto 0);
signal S: std_logic_vector(15 downto 0);
signal done: std_logic := '0';

constant PERIOD: time := 50 ns;

for UUT: fib use entity work.fib(behavior);          -- Configuration
                                                     -- specification
```

```
begin
  UUT: fib port map(Clk=>Clk,Clr=>Clr,Load=>Load,      -- Creates one
            Data_in=>Data_in,S=>S);                    -- instance

  Clock: process
    variable c: std_logic := '0';      -- Clock process, just like the last one
  begin
    while (done = '0') loop            -- The done flag indicates that we
      wait for PERIOD/2;               -- are finished and can stop the clock.
      c := not c;
      Clk <= c;
    end loop;
  end process;

  -- Process 'read_input' loops through all the lines in the file and
  -- extracts the test vector data. This process makes use of functions
  -- in the text I/O library (provided with IEEE Standard 1076) as well as
  -- the two conversion functions declared earlier in this architecture.
  -- Each line of the test vector file contains two fields: the input vector
  -- and the expected output values.
  Read_input: process
    file vector_file: text is in "testfib.vec";      -- Declare and open the
                                                     -- file (1076-1987 style).
    variable stimulus_in: std_logic_vector(33 downto 0); -- Inputs
    variable S_expected: std_logic_vector(15 downto 0); -- Outputs
    variable str_stimulus_in: string(34 downto 1);   -- The vector string
    variable err_cnt: integer := 0;                  -- Error counter
    variable file_line: line;                        -- Text line buffer; 'line' is a
                                                     -- standard type (textio library).
  begin
    wait until rising_edge(Clk);          -- Synchronize with first clock
    while not endfile(vector_file) loop   -- Loop through lines in the file

      readline (vector_file,file_line);   -- Read one complete line
                                          -- into file_line.
      read (file_line,str_stimulus_in) ;  -- Extract the first field from
                                          -- file_line.
      stimulus_in := str2vec(str_stimulus_in);   -- Convert the input
                                                 -- string to a vector

      wait for 1 ns;                      -- Delay for a nanosecond

      Clr <= stimulus_in(33);                    -- Get each input's
      Load <= stimulus_in(32);                   -- value from the test
      Data_in <= stimulus_in(31 downto 16);      -- vector array and
                                                 -- assigns the values
```

```
    -- Put the output side (expected values) into a variable:
        S_expected := stimulus_in(15 downto 0);

        wait until falling_edge(Clk);          -- Wait until the clock goes
                                               -- back to '0' (midway through
                                               -- the clock cycle)

    -- Check the expected value against the results in S:
        if (S /= S_expected) then
          err_cnt := err_cnt + 1;      -- Increment the error counter and
          assert false                 -- report an error if different
            report "Vector failure!" & lf &
            "Expected S to be  " & vec2str(S_expected) & lf &
            "but its value was " & vec2str(S) & lf
            severity note;
        end if;
      end loop;                        -- Continue looping through the file

      done <= '1';                     -- Set a flag when we are finished; this
                                       -- will stop the clock.

      wait;                            -- Suspend the simulation

    end process;

end stimulus;
```

Using file I/O for test data can reduce the time required to add or modify test data.

This test bench reads files of text "dynamically" during simulation, so the test bench does not have to be recompiled when test stimulus is added or modified. This is a big advantage for very large designs.

What does the test vector file that this test bench reads look like? The following file (**testfib.vec**) describes one possible sequence of tests that could be performed using this test bench:

```
10000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000101
00000000000000000000000000001000
```

```
00000000000000000000000000001101
00000000000000000000000000010101
00000000000000000000000000100010
00000000000000000000000000110111
00000000000000000000000001011001
00000000000000000000000010010000
00000000000000000000000011101001
00000000000000000000000101111001
00000000000000000000001001100010
00000000000000000000001111011011
00000000000000000000011000111101
00000000000000000000101000011000
00000000000000000001000001010101
00000000000000000001101001101101
00000000000000000010101011000010
00000000000000000100010100101111
00000000000000000110111111110001
00000000000000001011010100100000
00000000000000000010010100010001
00000000000000000000000000000001
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000101
00000000000000000000000000001000
```

This file could have been entered manually, using a text editor. Alternatively, it could have been generated from some other software package or from a program written in C, Basic or any other language. Reading text from files opens many new possibilies for testing and for creating interfaces between different design tools.

Test data stored in files can be modified with any text editor, and no recompile is required when the data are changed.

Although test vectors are quite useful for tabular test data, they are not particularly readable. In the last example of this chapter, we will describe how you can read and process test stimulus files that are more command-oriented, rather than simply being tables of binary values.

VHDL 1076-1993 File I/O

Before leaving the above example, let's see how it might look when coded using the 1076-1993 language features. In this version of the same test bench, we have modified the VHDL

source file to reflect changes in the 1076-1993 standard. In addition to adding various syntax enhancements (such as allowing an **is** keyword to be used in a component declaration) for readability and consistency, the 1993 specification adds additional features for better control over the opening and closing of files. In the following source file, we use the **file_open** built-in function to open the test vector file.

```
------------------------------------------------------
-- Test bench, VHDL '93 style
--
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.fib;    -- Get the design out of library 'work'

entity testfib is
end entity testfib;

architecture stimulus of testfib is
  component fib is
    port (Clk,Clr: in std_logic;
          Load: in std_ulogic;
          Data_in: in std_ulogic_vector(15 downto 0);
          S: out std_ulogic_vector(15 downto 0));
  end component fib;

  function str_to_stdvec(inp: string) return std_ulogic_vector is
    variable temp: std_ulogic_vector(inp'range) := (others => 'X');
  begin
    for i in inp'range loop
      if (inp(i) = '1') then
        temp(i) := '1';
      elsif (inp(i) = '0') then
        temp(i) := '0';
      end if;
    end loop;
    return temp;
  end function str_to_stdvec;

  function stdvec_to_str(inp: std_ulogic_vector) return string is
    variable temp: string(inp'left+1 downto 1) := (others => 'X');
  begin
    for i in inp'reverse_range loop
      if (inp(i) = '1') then
```

```
            temp(i+1) := '1';
          elsif (inp(i) = '0') then
            temp(i+1) := '0';
          end if;
        end loop;
        return temp;
      end function stdvec_to_str;

      signal Clk,Clr: std_ulogic;
      signal Load: std_ulogic;
      signal Data_in: std_ulogic_vector(15 downto 0);
      signal S: std_ulogic_vector(15 downto 0);
      signal done: std_ulogic := '0';

      constant PERIOD: time := 50 ns;

    begin
      UUT: fib port map(Clk=>Clk,Clr=>Clr,Load=>Load,
                  Data_in=>Data_in,S=>S);

      Clock: process
        variable c: std_ulogic := '0';
      begin
        while (done = '0') loop
          wait for PERIOD/2;
          c := not c;
          Clk <= c;
        end loop;
      end process Clock;

      Read_input: process
        file vector_file: text;

        variable stimulus_in: std_ulogic_vector(33 downto 0);
        variable S_expected: std_ulogic_vector(15 downto 0);
        variable str_stimulus_in: string(34 downto 1);
        variable err_cnt: integer := 0;
        variable file_line: line;

      begin

        file_open(vector_file,"tfib93.vec",READ_MODE);

        wait until rising_edge(Clk);

        while not endfile(vector_file) loop
```

```
        readline (vector_file,file_line);
        read (file_line,str_stimulus_in) ;
        assert (false)
          report "Vector: " & str_stimulus_in
          severity note;
        stimulus_in := str_to_stdvec (str_stimulus_in);

        wait for 1 ns;

        --Get input side of vector...
        Clr <= stimulus_in(33);
        Load <= stimulus_in(32);
        Data_in <= stimulus_in(31 downto 16);

        --Put output side (expected values) into a variable...
        S_expected := stimulus_in(15 downto 0);

        wait until falling_edge(Clk);

        -- Check the expected value against the results...
        if (S /= S_expected) then
          err_cnt := err_cnt + 1;
          assert false
            report "Vector failure!" & lf &
            "Expected S to be  " & stdvec_to_str(S_expected) & lf &
            "but its value was " & stdvec_to_str(S) & lf
            severity note;
        end if;
      end loop;

      file_close(vector_file);

      done <= '1';

      if (err_cnt = 0) then
        assert false
          report "No errors." & lf & lf
          severity note;
      else
        assert false
          report "There were errors in the test." & lf
          severity note;
      end if;
      wait;

    end process Read_input;
```

```
end architecture stimulus;

-- Add a configation statement. This statement actually states the
-- default configuration, and so it is optional.
configuration build1 of testfib is
  for stimulus
    for DUT: fib use entity work.fib(behavior)
      port map(Clk=>Clk,Clr=>Clr,Load=>Load,
            Data_in=>Data_in,S=>S);
    end for;
  end for;
end configuration build1;
```

Reading Non-tabular Data from Files

You can use VHDL's text I/O features to read and write many different built-in data types, including such data types as characters, strings, and integers. This is a powerful feature of the language that you will make great use of as you become proficient with the language.

The standard text I/O features do not include functions for reading and writing standard logic data types.

VHDL's text I/O features are somewhat limited, however, when it comes to reading data that is not expressed as one of the built-in types defined in Standard 1076. The primary example of this is when you wish to read or write standard logic data types. In the previous example (the Fibonacci sequence generator), we made use of type coversion functions to read standard logic input data as characters. This method works fine, but it is somewhat clumsy. A better way to approached this common problem is to develop a reusable package of functions for reading and writing standard logic data. Writing a comprehensive package of such functions is not a trivial task. It would probably require a few days of coding and debugging.

Fortunately, one such package already exists and is in widespread use. This package, **std_logic_textio**, was originally developed by Synopsys. Synopsys allows the package to be used and distributed without restriction. We will use the

std_logic_textio package to demonstrate how you might read data fields from a file and write other data to another file (or, in this case, to the console or simulator transcript window).

The circuit that we will be testing with our test bench is a 32-bit adder-subtractor unit, the complete source code for which is provided on the companion CD-ROM. The test bench that we wish to write will read information from a file in the form of hexadecimal numeric values. The data file, which we will name TST_ADD.DAT, will include both the inputs and the expected outputs for the circuit. A listing of TST_ADD.DAT, containing a small number of test lines, is shown below:

Special type conversion functions are required to read and write data in hexadecimal format.

```
0 00000001 00000001 00000002 0
0 00000002 00000002 00000004 0
0 00000004 00000004 00000008 0
0 FFFFFFFF FFFFFFFF FFFFFFFE 1
0 0000AAAA AAAA0000 AAAAAAAA 0
0 158D7129 E4C28B56 FA4FFC7F 0
1 00000001 00000001 00000000 0
1 A4F67B92 00000001 5B09846F 0
1 FFFFFFFF FFFFFFFE FFFFFFFF 0
1 FFFFFFFE FFFFFFFF 00000001 1
1 00000002 00000004 00000002 1
```

The standard text I/O features defined in VHDL standard 1076 do not include procedures to read data in hexadecimal format, so we will make use of the **hread** procedure provided in the Synopsys **std_logic_textio** package. **Hread** accepts the same arguments as the standard **read** procedure, but allows values to be expressed in hexadecimal format. We will use **hread** to read the second, third and fourth fields in each line of the file, as these fields are represented in hexadecimal format.

Because the first and last fields of the data file are single-bit values of type **std_ulogic**, we will also make use of an overloaded **read** procedure provided in **std_logic_textio**. VHDL's built-in **read** procedure is not capable of reading **std_ulogic** values, so the **std_logic_textio** package includes additional **read** procedure definitions that extend **read** for these values.

Finally, we wish to display the results of simulation in the simulator's transcript window, so we use the **hwrite** and overloaded **write** procedures provided in **std_logic_textio** to format and display the data values. Once again, these are procedures that are not provided in the standard VHDL text I/O package.

*The **std_logic_textio** package provided by Synopsys helps when reading standard logic data values from files.*

```
library ieee;
use ieee.std_logic_1164.all;
use work.all;
use std.textio.all;

library textutil;      -- Synposys Text I/O package
use textutil.std_logic_textio.all;

entity tst_add is
end tst_add;

architecture readhex of tst_add is
  component adder32 is
    port (cin: in std_ulogic;
          a,b: in std_ulogic_vector(31 downto 0);
          sum: out std_ulogic_vector(31 downto 0);
          cout: out std_ulogic);
  end component;
  for all: adder32 use entity work.adder32(structural);
  signal Clk: std_ulogic;
  signal x, y: std_ulogic_vector(31 downto 0);
  signal sum: std_ulogic_vector(31 downto 0);
  signal cin, cout: std_ulogic;

  constant PERIOD: time := 200 ns;

begin
  UUT: adder32 port map (cin, x, y, sum, cout);

  readcmd: process

    -- This process loops through a file and reads one line
    -- at a time, parsing the line to get the values and
    -- expected result.
    --
    -- The file format is CI A B SUM CO, with A, B and SUM
    -- expressed as hexadecimal values.
```

```
  file cmdfile: TEXT;      -- Define the file 'handle'
  variable line_in,line_out: Line; -- Line buffers
  variable good: boolean;  -- Status of the read operations

  variable CI, CO: std_ulogic;
  variable A,B: std_ulogic_vector(31 downto 0);
  variable S: std_ulogic_vector(31 downto 0);
  constant TEST_PASSED: string := "Test passed:";
  constant TEST_FAILED: string := "Test FAILED:";

  -- Use a procedure to generate one clock cycle...
  procedure cycle (n: in integer) is
  begin
    for i in 1 to n loop
      Clk <= '0';
      wait for PERIOD / 2;
      Clk <= '1';
      wait for PERIOD / 2;
    end loop;
  end cycle;

begin

  -- Open the command file...

  FILE_OPEN(cmdfile,"TST_ADD.DAT",READ_MODE);

  loop

    if endfile(cmdfile) then  -- Check EOF
      assert false
        report "End of file encountered; exiting."
        severity NOTE;
      exit;
    end if;

    readline(cmdfile,line_in);    -- Read a line from the file
    next when line_in'length = 0;  -- Skip empty lines

    read(line_in,CI,good);    -- Read the CI input
    assert good
      report "Text I/O read error"
      severity ERROR;

    hread(line_in,A,good);    -- Read the A argument as hex value
    assert good
```

```
      report "Text I/O read error"
      severity ERROR;

    hread(line_in,B,good);    -- Read the B argument
    assert good
      report "Text I/O read error"
      severity ERROR;

    hread(line_in,S,good);    -- Read the Sum expected resulted
    assert good
      report "Text I/O read error"
      severity ERROR;

      read(line_in,CO,good);    -- Read the CO expected resulted
    assert good
      report "Text I/O read error"
      severity ERROR;

    cin <= CI;
    x <= A;
    y <= B;

    wait for PERIOD;  -- Give the circuit time to stabilize

    if (sum = S) then
      write(line_out,TEST_PASSED);
    else
      write(line_out,TEST_FAILED);
    end if;
    write(line_out,CI,RIGHT,2);
    hwrite(line_out,A,RIGHT,9);
    hwrite(line_out,B,RIGHT,9);
    hwrite(line_out,sum,RIGHT,9);
    write(line_out,cout,RIGHT,2);
    writeline(OUTPUT,line_out);    -- write the message

  end loop;

  wait;

end process;

end architecture readhex;
```

*Overloaded **read** and **hread** procedures accept arguments of type **std_logic_vector**.*

Creating a Test Language

As you can see,VHDL is a robust programming language with features that go well beyond what is needed to describe logic. In this, our final example, we will show how you can use the features of VHDL to describe a more complex test bench that reads non-tabular, command-oriented information from a text file, parses that information to determine the appropriate test inputs, and performs output value checking as specified in the file. This example will make use of a number of built-in functions for file and text manipulation, as well as text I/O functions for the parsing and display of a variety of different data types, including strings, characters and numeric values.

VHDL is a robust programming language with features that go well beyond what is needed to describe logic...

We'll use the language features of VHDL 1076-1993 for this test bench example, although the features of VHDL 1076-1987 could also be used to perform the same tasks, with some minor modifications.

The design we will be testing is a driving game that was inspired by the "ChipTrip" example first described by Altera Corporation using their AHDL PLD language. In our version of the design (which is described in more detail in *Electronic Design Automation for Windows: a User's Guide*, published in 1995 by Prentice Hall), the objective is to create a sequence of test inputs that will cause an imaginary work-weary engineer to proceed from his office to the beach, as quickly as possible, without getting a speeding ticket. To make the trip more interesting, our hero must stop and pick up a pizza on the way. The map of Figure 9-2 illustrates the possible routes that can be taken.

This map shows three different types of roads: freeways, commercial streets, and residential roads. The car being driven has only two possible speeds, fast and slow. When the car is driven slowly, it advances from one point on the map (say, from **Ramp1** to **Ramp2**) in a given period of time. When driven fast, the car proceeds twice as far. There is no speed limit on the freeway, so the car can travel at full speed without fear of getting a ticket. On commercial streets, the car may

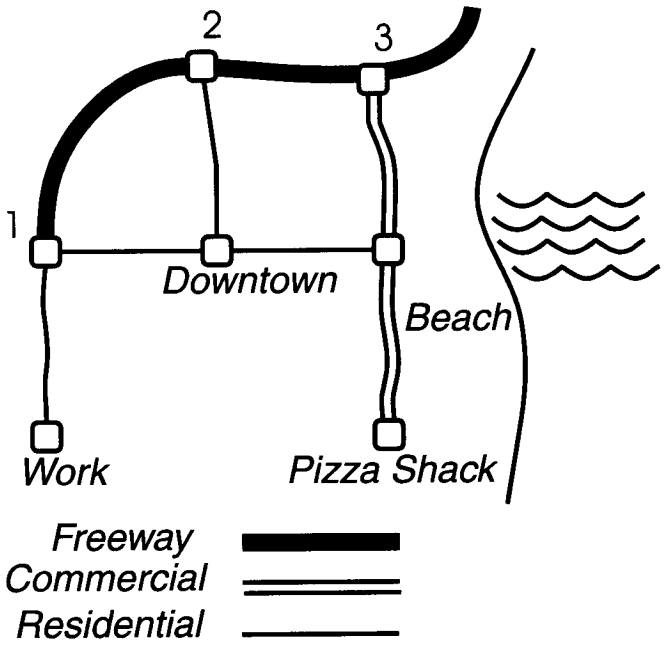

***Figure 9-2**: The driving game (inspired from Altera's ChipTrip design example) simulates a drive across town. The goal: get from the office to the beach (and pick up a pizza on the way) in the shortest amount of time without getting a ticket. Surf's up!*

exceed the speed limit just once and get away with it. On residential roads, any attempt to drive fast will result in a ticket.

In our simulation, and in the underlying design description, a fixed period of time is represented by a single clock cycle. Inputs for the speed and initial direction of travel are represented by signals **Speed** and **Dir**. The location of the car at any point is represented internally to the circuit by a state machine, but it is kept hidden at the top level of the design and in the test bench itself. The current status and success or failure of a trip are observed on the signals **DriveTime**, **Tickets**, and **Party**, which tell the player how long the drive has taken, how many traffic tickets have accrued, and whether he or she has yet arrived at the beach with the pizza. The block diagram of Figure 9-3 shows the general layout of the elements of this

design, which consists of two state machines, two counters and some additional logic. (The VHDL source files for the entire design are listed in Appendix F.)

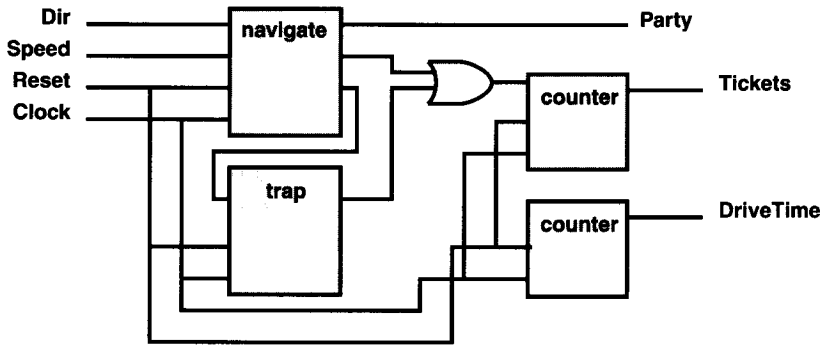

Figure 9-3: The driving game is composed of two state machines, a counter (repeated twice) and some random logic.

Test Bench Requirements

The test bench that we have written for this design reads symbolic test commands from a file, allowing the game to be easily tested and various driving scenarios to be described. There are three basic commands allowed in the test input file: **RESET**, **DRIVE** and **CHECK**.

The **RESET** command causes the game to be reset to its initial state. When this command is encountered in the input file, the **DriveTime** and **Tickets** counters are reset to zero, and the state machine that controls the drive is internally reset to state **Office**.

The **DRIVE** command has two additional arguments for direction and speed. It also specifies where the car should go, and how far, in one clock cycle. When **DRIVE** is encountered in the input file, the direction and speed arguments are parsed, the appropriate inputs (**Dir** and **Speed**) are assigned, and a single clock pulse is generated to advance the car to the next symbolic location on the map. In the test input file, the

direction is represented by the values **NORTH, SOUTH, EAST** and **WEST**, while the speed is represented by the values **FAST** and **SLOW**.

The **CHECK** command allows the value of signal **Party** or signal **Tickets** to be checked at any time to verify that the drive has proceeded as expected, and to allow the design to be tested for various sequences of directions and speeds.

Note:

To simplify the text field parsing in this example, we have intentionally made all commands read from the file the same size, in terms of the number of characters read for a given keyword. This includes the keywords "EAST " and "WEST ", "FAST " and "SLOW ", which in this implementation have an extra space appended onto them. To read string data (keywords) of random size, we would have to write a more complex token parser that reads the input line one character at a time, looking forward one character to identify delimiters such as spaces, tabs, and so on. This is not a difficult thing to do in VHDL, but it would clutter up an otherwise concise example.

The following sequence of test commands, entered into a file named **Tstpizza.cmd,** describes a number of possible test sequences for this game. Notice that the test file includes comments fields which are ignored by the test bench when the file is read:

The command language we have defined allows comment lines, and uses English-like keywords to describe the test sequence.

```
-- Test command file for getpizza example.
-- First try a winning game...
--
RESET
DRIVE NORTH SLOW      -- Cruise from work to Ramp 1
DRIVE NORTH FAST      -- No speed limit here!
DRIVE SOUTH FAST      -- Speed along the beach (1 warning)
DRIVE NORTH SLOW      -- Got the pizza; back to the beach!
CHECK PARTY 1
CHECK TICKS 0
--
-- Try getting some tickets this time...
--
RESET
DRIVE NORTH FAST      -- Speed through the residential area
DRIVE SOUTH SLOW      -- Cruise south to downtown
DRIVE EAST SLOW       -- Cruise to the beach
```

```
DRIVE WEST FAST        -- Speed back through town
DRIVE EAST FAST        -- Speed back to the beach
DRIVE SOUTH SLOW       -- Go get the pizza
DRIVE NORTH SLOW       -- Party time!
CHECK PARTY 1
CHECK TICKS 3
```

Test Bench Source File

The following source file, with explanatory comments, reads in the test file as described above and applies the necessary input stimuli to make the game proceed. **Assert** statements are used to verify that the test commands are properly entered, and to display errors when the **CHECK** command indicates a value for **Tickets** or **Party** other than those calculated during simulation.

```
------------------------------------------------------------
-- This test bench tests the getpizza driving game. It is
-- written with 1076-1993 features, and it makes use of
-- IEEE 1076.3 and the standard text I/O packages.
--
-- The test bench reads commands from a file (tstpizza.cmd)
-- to determine how the game should be "played".
--

use std.textio.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;   -- This design uses 'unsigned'

use work.game_types.all;  -- Contains types and constants

use work.pizzatop;

entity testgame is
end entity testgame;

architecture parser of testgame is
  component pizzatop is
    port(Clk,Reset: in std_logic;
         Speed: in tSpeed;
         Dir: in tDirection;
         Party: out std_logic;
```

```
        Tickets: out std_logic_vector(3 downto 0);
        DriveTime: out std_logic_vector(3 downto 0));
    end component pizzatop;

    signal Reset,Clk: std_logic;
    signal Dir: tDirection;    -- std_logic_vector(1 to 2)
    signal Speed: tSpeed;    -- std_logic
    signal Party: std_logic;
    signal DriveTime, Tickets: std_logic_vector(3 downto 0);

    constant PERIOD: time := 40 ns;

begin

    UUT: pizzatop port map (Clk,Reset,Speed,Dir,Party,Tickets,DriveTime);

    readcmd: process

        -- This process loops through a file and reads one line
        -- at a time, parsing the line to get the commands.

        file cmdfile: TEXT;       -- Define the file 'handle'
        variable L: Line;         -- Define the line buffer

        variable keyword: string (1 to 5);  -- Used to get a keyword
        variable c: character;    -- Used to read a single character
        variable value: integer;  -- Used to read a numeric value
        variable good: boolean;   -- For optional text I/O error checks

        -- Use a procedure to generate one clock cycle...
        procedure cycle (n: in integer) is
        begin
          for i in 1 to n loop
            Clk <= '0';
            wait for PERIOD / 2;
            Clk <= '1';
            wait for PERIOD / 2;
          end loop;
        end cycle;

    begin

        -- Open the command file...

        file_open(cmdfile,"tstpizza.cmd",READ_MODE);
```

```
LOOP1: loop

  if endfile(cmdfile) then  -- Check EOF
    assert false
      report "End of file encountered; exiting."
      severity NOTE;
    exit LOOP1;
  end if;

  readline(cmdfile,L);          -- Read the line
  next LOOP1 when L'length = 0;  -- Skip empty lines

  read(L,keyword,good);      -- Read the command keyword
  assert good
    report "Text I/O read error"
    severity ERROR;
  case keyword is            -- Parse the command...
    when "RESET" =>
      Reset <= '1';
      cycle(1);
      Reset <= '0';
    when "DRIVE" =>
      read(L,c);         -- Eat the white space
      read(L,keyword);   -- Read the direction
      case keyword is
        when "NORTH" =>
          Dir <= North;
        when "SOUTH" =>
          Dir <= South;
        when "EAST " =>  -- Note the extra space
          Dir <= East;
        when "WEST " =>
          Dir <= West;
        when others =>
          assert false
            report "Unknown direction"
            severity ERROR;
      end case;
      read(L,c);         -- Eat the white space
      read(L,keyword);   -- Read the speed
      case keyword is
        when "SLOW " =>
          Speed <= SLOW;
        when "FAST " =>
          Speed <= FAST;
        when others =>
```

```
                    assert false
                      report "Unknown speed"
                      severity ERROR;
                end case;
                cycle(1);
              when "CHECK" =>
                read(L,c);        -- Eat the white space
                read(L,keyword);  -- Read the signal word
                case keyword is
                  when "PARTY" =>
                    read(L,value);   -- Get a value
                    if value = 0 then
                      assert (Party = '0')
                        report "Check failed on Party!"
                        severity ERROR;
                    else
                      assert (Party = '1')
                        report "Check failed on Party!"
                        severity ERROR;
                    end if;
                  when "TICKS" =>
                    read(L,value);   -- Get a value
                    assert (UNSIGNED(Tickets) = value)
                      report "Check failed on Ticket count!"
                      severity ERROR;
                  when others =>
                    null;
                end case;
              when others =>     -- Only comments are valid here...
                assert keyword(1 to 2) = "--"  -- Comment?
                  report "Unknown keyword"
                  severity ERROR;
            end case;

          end loop LOOP1;

          wait;

        end process;

      end architecture parser;
```