XAPP199 (v1.1) May 17, 2010

Writing Efficient Testbenches

Author: Mujtaba Hamid

Summary

This application note is written for logic designers who are new to HDL verification flows, and who do not have extensive testbench-writing experience.

Testbenches are the primary means of verifying HDL designs. This application note provides guidelines for laying out and constructing efficient testbenches. It also provides an algorithm to develop a self-checking testbench for any design.

All design files for this application note are available on the FTP site at:

PC: **https://secure.xilinx.com/webreg/clickthrough.do?cid=134003&license=RefDesLicense**

UNIX:
https://secure.xilinx.com/webreg/clickthrough.do?cid=134002&license=RefDesLicense

Introduction

Due to increases in design size and complexity, digital design verification has become an increasingly difficult and laborious task. To meet this challenge, verification engineers rely on several verification tools and methods. For large, multi-million gate designs, engineers typically use a suite of formal verification tools. However, for smaller designs, design engineers usually find that HDL simulators with testbenches work best.

Testbenches have become the standard method to verify HLL (High-Level Language) designs. Typically, testbenches perform the following tasks:

- Instantiate the design under test (DUT)
- Stimulate the DUT by applying test vectors to the model
- Output results to a terminal or waveform window for visual inspection
- Optionally compare actual results to expected results

Typically, testbenches are written in the industry-standard VHDL or Verilog hardware description languages. Testbenches invoke the functional design, then stimulate it. Complex testbenches perform additional functions—for example, they contain logic to determine the proper design stimulus for the design or to compare actual to expected results.

The remaining sections of this note describe the structure of a well-composed testbench, and provide an example of a self-checking testbench—one that automates the comparison of actual to expected testbench results.

Figure 1 shows a standard HDL verification flow which follows the steps outlined above.

Since testbenches are written in VHDL or Verilog, testbench verification flows can be ported across platforms and vendor tools. Also, since VHDL and Verilog are standard non-proprietary languages, verification suites written in VHDL or Verilog can be reused in future designs without difficulty.

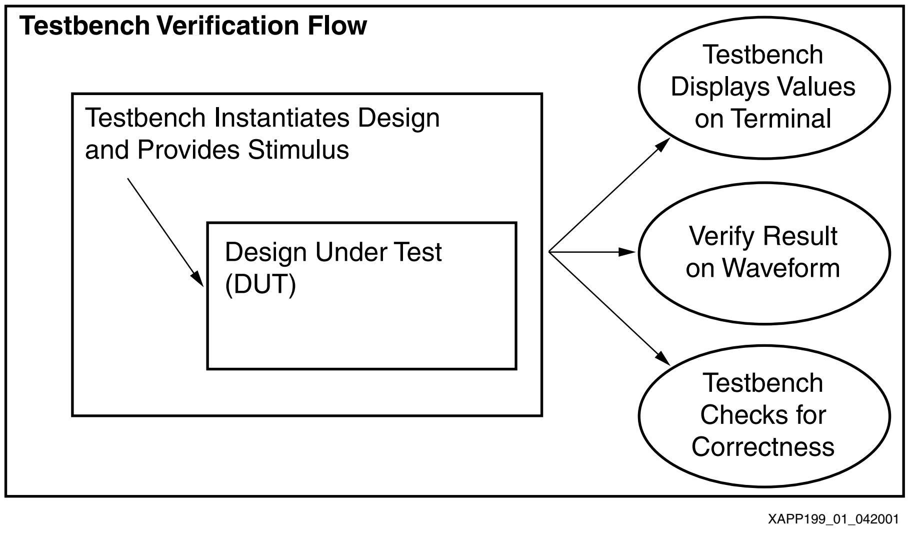

Testbench Verification Flow
Testbench Instantiates Design and Provides Stimulus
Design Under Test (DUT)
Testbench Displays Values on Terminal
Verify Result on Waveform
Testbench Checks for Correctness
XAPP199_01_042001

Figure 1: **HDL Verification Flow Using Testbenches**

Constructing Testbenches

Testbenches can be written in VHDL or Verilog. Since testbenches are used for simulation only, they are not limited by semantic constraints that apply to RTL language subsets used in synthesis. Instead, all behavioral constructs can be used. Thus, testbenches can be written more generically, making them easier to maintain.

All testbenches contain the basic sections shown in Table 1. As mentioned, above, testbenches typically contain additional functionality as well, such as the visual display of results on a terminal and built-in error detection.

Table 1: **Sections Common to Testbenches**

VHDL	Verilog
Entity and Architecture Declaration	Module Declaration
Signal Declaration	Signal Declaration
Instantiation of Top-level Design	Instantiation of Top-level Design
Provide Stimulus	Provide Stimulus

The following examples show some constructs used frequently in testbenches.:

Generating Clock Signals

Designs that use system clocks to sequence logic must generate a clock. Iterative clocks can easily be implemented in both VHDL and Verilog source code. The following are VHDL and Verilog examples of clock generation:

VHDL:

```
-- Declare a clock period constant.
Constant ClockPeriod : TIME := 10 ns;
-- Clock Generation method 1:
Clock <= not Clock after ClockPeriod / 2;
-- Clock Generation method 2:
GENERATE_CLOCK: process
begin
```

```
wait for (ClockPeriod / 2)
Clock <= ’1’;
wait for (ClockPeriod / 2)
Clock <= ’0’;
end process;
```

Verilog:

```
// Declare a clock period constant.
Parameter ClockPeriod = 10;
// Clock Generation method 1:
initial begin
forever Clock = #(ClockPeriod / 2) ~ Clock;
end
// Clock Generation method 2:
initial begin
always #(ClockPeriod / 2) Clock = ~Clock;
end
```

Providing Stimulus

To obtain testbench verification results, stimulus must be provided to the DUT. Concurrent stimulus blocks are used in testbenches to provide the necessary stimuli. Two methods are employed: absolute-time stimulus and relative-time stimulus. In the first method, simulation values are specified relative to simulation time zero. By comparison, relative-time stimulus supplies initial values, then waits for an event before retriggering the stimulus. Both methods can be combined in a testbench, according to the designer’s needs.

Table 2 and Table 3 provide examples of absolute-time and relative-time stimuli, respectively, in VHDL and Verilog source code.

Table 2: **Absolute Time Stimulus Example**

VHDL-ABSOLUTE TIME	Verilog-ABSOLUTE TIME

`MainStimulus: process begin`
`Reset <= '1';`
`Load <= '0';`
`Count_UpDn <= '0';`
`wait for 100 ns;`
`Reset <= '0';`
`wait for 20 ns;`
`Load <= '1';`
`wait for 20 ns;`
`Count_UpDn <= '1';`
`end process;`
	`initial begin`
	`Reset = 1;`
	`Load = 0;`
	`Count_UpDn = 0;`
	`#100 Reset = 0;`
	`#20 Load = 1;`
	`#20 Count_UpDn = 1;`
	`end`

Table 3: **Relative Time Stimulus Example**

VHDL-RELATIVE TIME	Verilog-RELATIVE TIME
`Process (Clock)`	`always @ (posedge clock)`
`Begin`	`TB_Count <= TB_Count + 1;`
`If rising_edge(Clock) then`	`initial begin`
`TB_Count <= TB_Count + 1;`	`if (TB_Count <= 5)`
`end if;`	` begin`
`end process;`	` Reset = 1;`
`SecondStimulus: process begin`	` Load = 0;`
`if (TB_Count <= 5) then`	` Count _UpDn = 0;`
`Reset <= '1';`	` end`
`Load <= '0';`	`else`
`Count_UpDn <= '0';`	` begin`
`Else`	` Reset = 0;`
`Reset <= '0';`	` Load = 1;`
`Load <= ‘1’;`	` Count_UpDn = 1;`
`Count_UpDn <= ‘1’;`	` end`
`end process;`	`end`
`FinalStimulus: process begin`	`initial begin`
` if (Count = "1100") then`	` if (Count == 1100) begin`
`  Count_UpDn <= '0';`	`  Count_UpDn <= 0;`
`  report "Terminal Count`	`  $display("Terminal Count`
`Reached, now counting down."`	`Reached, now counting down.");`
` end if;`	`  end`
`end process;`	`end`

VHDL process blocks and Verilog initial blocks are executed concurrently along with other process and initial blocks in the file. However, within each (process or initial) block, events are scheduled sequentially, in the order written. This means that stimulus sequences begin in each concurrent block at simulation time zero. Multiple blocks should be used to break up complex stimulus sequences into more readable and maintainable code.

Displaying Results

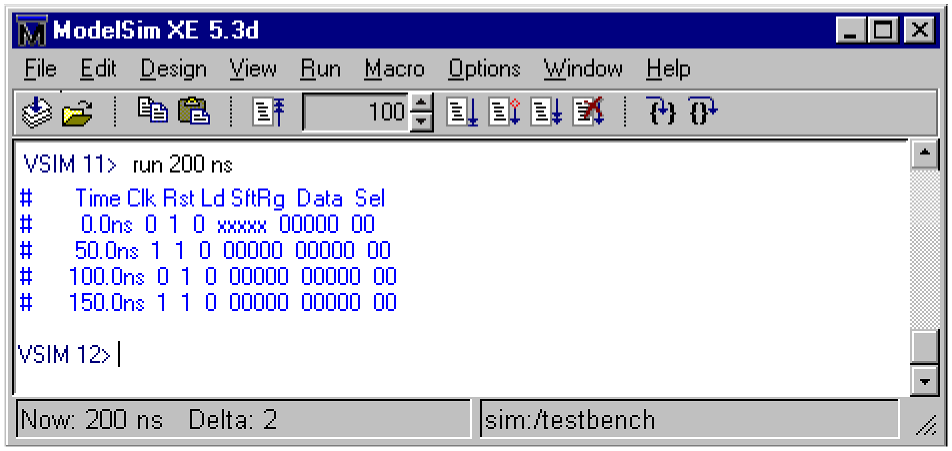
Displaying results is facilitated in Verilog by the $display and $monitor keywords. Although VHDL does not have equivalent display-specific commands, it provides the std_textio package, which allows file I/O redirection to the display terminal window (for an example of this technique, see Self-Checking Testbenches, below).

The following is a Verilog example in which values are displayed on the terminal screen:

```
// pipes the ASCII results to the terminal or text editor
initial begin
  $timeformat(-9,1,"ns",12);
  $display("   Time Clk Rst Ld SftRg Data Sel");
  $monitor("%t %b %b %b %b %b %b", $realtime,
       clock, reset, load, shiftreg, data, sel);
 end
```

The $display keyword outputs quoted parenthetical text (“...”) to the terminal window. The $monitor keyword works differently, since its output is event-driven. In the example, the $realtime variable (assigned by the user to the current simulation time) is used to trigger the display of values in the signal list. The signal list starts with the $realtime variable, and is followed by the names of other signals whose values are to be displayed (clock, reset, load, and others). The beginning “%” keywords comprise a list of format specifiers, used to control how each signal value in the signal list is formatted for display. The format list is positional—each format specifier is sequentially associated with a successive signal name in the signal list. For example, the %t specifier formats the displayed $realtime value in time format, and the first %b specifier formats the clock value in binary format. Verilog provides additional format-specifiers, for example, %h is used for hexadecimal, %d for decimal, and %o for octal formats (consult a Verilog reference for a complete list of keywords and format specifiers).

The formatted display results are shown in Figure 2.



```
VSIM 11> run 200 ns
#     Time Clk Rst Ld SftRg Data Sel
#      0.0ns 0 1 0 xxxxx 00000 00
#     50.0ns 1 1 0 00000 00000 00
#    100.0ns 0 1 0 00000 00000 00
#    150.0ns 1 1 0 00000 00000 00

VSIM 12>
```

```
Now: 200 ns   Delta: 2    sim:/testbench
```

Figure 2: **Simulation Results Echoed to Terminal**

Simple Testbenches

Simple testbenches instantiate the user design, then provide stimuli to it. Testbench output is displayed graphically on the simulator's waveform window or as text sent to the user's terminal or to a piped text file.

Below is a simple Verilog design representing a shift register:

```
module shift_reg (clock, reset, load, sel, data, shiftreg);
input clock;
input reset;
input load;
input [1:0] sel;
input [4:0] data;
output [4:0] shiftreg;
reg [4:0] shiftreg;
always @ (posedge clock)
begin
 if (reset)
   shiftreg = 0;
 else if (load)
   shiftreg = data;
 else
  case (sel)
   2'b00 : shiftreg = shiftreg;
   2'b01 : shiftreg = shiftreg << 1;
   2'b10 : shiftreg = shiftreg >> 1;
   default : shiftreg = shiftreg;
  endcase
end
endmodule
```

The following simple testbench examples instantiate the shift register design.

Verilog Example:

```
 module testbench; // declare testbench name
  reg clock;
  reg load;
```

```
reg reset;  // declaration of signals
wire [4:0] shiftreg;
reg [4:0] data;
reg [1:0] sel;
 // instantiation of the shift_reg design below
shift_reg dut(.clock (clock),
             .load (load),
             .reset (reset),
             .shiftreg (shiftreg),
             .data (data),
             .sel (sel));
 //this process block sets up the free running clock
initial begin
clock = 0;
forever #50 clock = ~clock;
end
initial begin// this process block specifies the stimulus.
  reset = 1;
  data = 5'b00000;
  load = 0;
  sel = 2'b00;
 #200
  reset = 0;
  load = 1;
 #200
  data = 5'b00001;
 #100
  sel = 2'b01;
  load = 0;
 #200
  sel = 2'b10;
 #1000 $stop;
end
initial begin// this process block pipes the ASCII results to the
//terminal or text editor
$timeformat(-9,1,"ns",12);
$display("   Time Clk Rst Ld SftRg Data Sel");
$monitor("%t %b %b %b %b %b %b", $realtime,
    clock, reset, load, shiftreg, data, sel);
end
endmodule
```

The testbench, above, instantiates the design, sets up the clock, then provides the stimuli. All process blocks start at simulation time zero and are concurrent. The pound sign (#) specifies the delay before the next stimulus is applied. The $stop command instructs the simulator to stop testbench simulation (all testbenches should contain a stop command). Finally, the $monitor statement echoes the results in ASCII format to the screen or a piped text editor.

Following is a VHDL testbench that instantiates and provides stimulus to the Verilog shift register design above.

VHDL Example:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity testbench is
end entity testbench;
architecture test_reg of testbench is
component shift_reg is
```

```
port (clock : in std_logic;
  reset : in std_logic;
  load : in std_logic;
  sel : in std_logic_vector(1 downto 0);
  data : in std_logic_vector(4 downto 0);
  shiftreg : out std_logic_vector(4 downto 0));
end component;
signal clock, reset, load: std_logic;
signal shiftreg, data: std_logic_vector(4 downto 0);
signal sel: std_logic_vector(1 downto 0);
constant ClockPeriod : TIME := 50 ns;
begin
UUT : shift_reg port map (clock => clock, reset => reset,
   load => load, data => data,
   shiftreg => shiftreg);
process begin
 clock <= not clock after (ClockPeriod / 2);
end process;
process begin
 reset <= '1';
 data <= "00000";
 load <= '0';
 set <= "00";
  wait for 200 ns;
 reset <= '0';
 load <= '1';
  wait for 200 ns;
 data <= "00001";
  wait for 100 ns;
 sel <= "01";
 load <= '0';
  wait for 200 ns;
 sel <= "10";
  wait for 1000 ns;
end process;
end architecture test_reg;
```

The VHDL testbench above is similar in functionality to the Verilog testbench shown previously, with the exception of a command to echo the output to the terminal. In VHDL, the std_textio package is used to display information to the terminal, and this will be covered in the next section.

Automatic Verification

Automating the verification of testbench results is recommended, particularly for larger designs. Automation reduces the time required to check a design for correctness, and minimizes human error.

Several methods are commonly used to automate testbench verification:

1. **Database Comparisons.** First, a database file containing expected output (a “golden vector” file) is created. Then, simulation outputs are captured and compared to the reference vectors in the golden vector file (the unix diff utility can be used to compare the ASCII database files). However, since pointers from output to input files are not provided, a disadvantage of this method is the difficulty of tracing an incorrect output to the source of the error.
2. **Waveform Comparison.** Waveform comparisons can be performed automatically or manually. The automatic method employs a testbench comparator to compare a golden waveform against the testbench output waveform. The Xilinx HDL Bencher tool can be

used to perform automatic waveform comparisons (for HDL Bencher information, go to: http://www.xilinx.com/products/software/statecad/index.htm)

3. **Self-Checking Testbenches.** A self-checking testbench checks expected results against actual results at run time, not at the end of simulation. Since useful error-tracking information can be built into the testbench to show where a design fails, debugging time is significantly shortened. Further information on self-checking testbenches is provided in the next section

Self-Checking Testbenches

Self-checking testbenches are implemented by placing a series of expected vectors in a testbench file. These vectors are compared at defined run-time intervals to actual simulation results. If actual results match expected results, the simulation succeeds. If results do not match expectations, the testbench reports the discrepancies.

Implementing self-checking testbenches is simpler for synchronous designs since expected and actual results can be compared at a clock edge or after every "n" clock cycles. Comparison methods also depend on the nature of the design. For example, a testbench for memory I/O should check results each time new data is written to or read from a memory location. Similarly, if a design uses a significant number of combinatorial blocks, combinatorial delays must be taken into account when expected results are specified.

In a self-checking testbench, expected outputs are compared to actual outputs at regular run-time intervals to provide automatic error checking. This technique works fine in small to mid-size designs. However, since possible output combinations increase exponentially with design complexity, writing a self-checking testbench for a large design is much more difficult and time consuming.

Below are examples of simple, self-checking testbenches written in Verilog and VHDL:

Verilog Example

Following the instantiation of the design, expected results are specified. Later in the code, expected and actual results are compared, and the results are echoed to the terminal. If there are no mismatches, an "end of good simulation" message is displayed. If a mismatch occurs, an error is reported along with the mismatched expected and actual values.

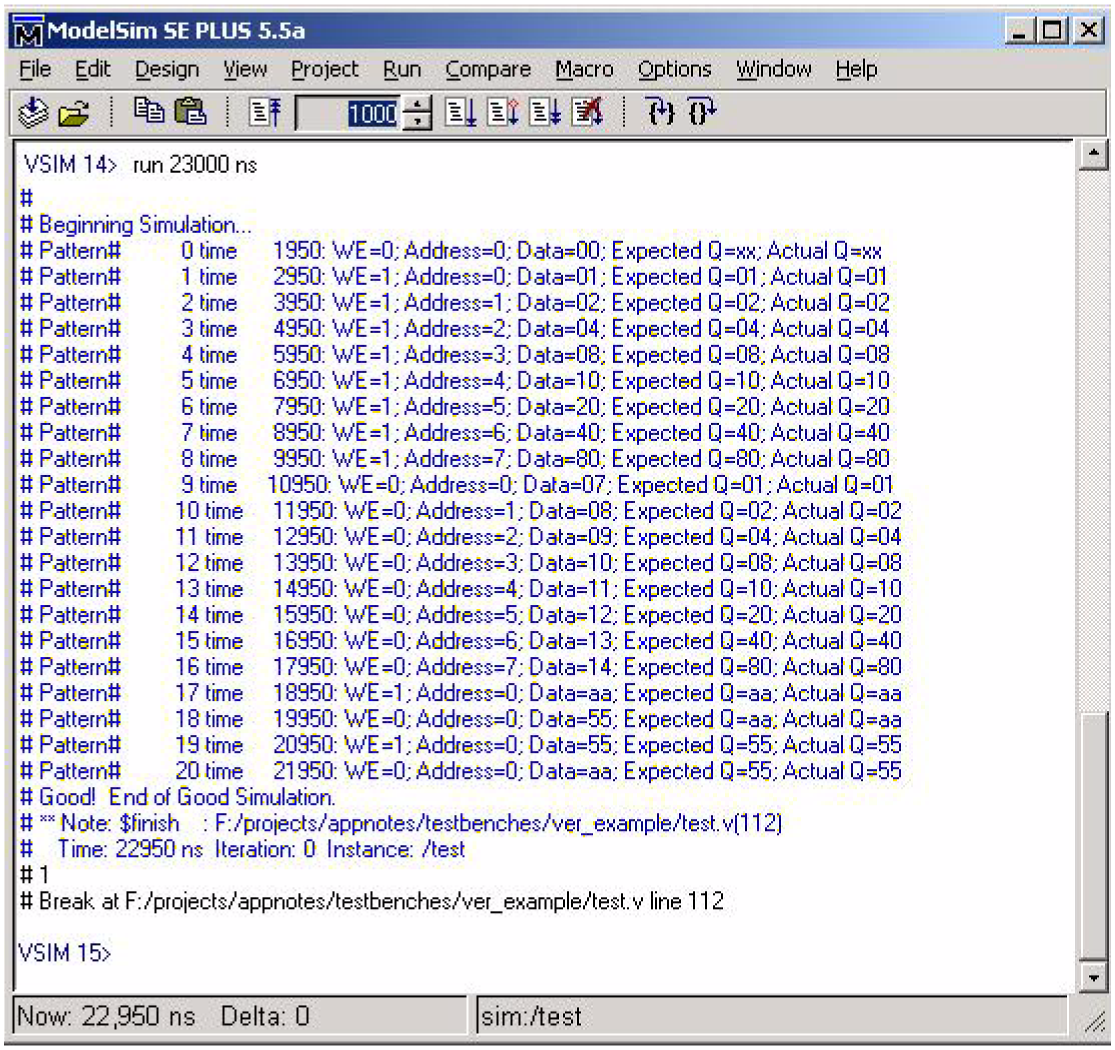
```
 `timescale 1 ns / 1 ps
module test_sc;
 reg tbreset, tbstrtstop;
 reg tbclk;
 wire [6:0] onesout, tensout;
 wire [9:0] tbtenthsout;
parameter cycles = 25;
reg [9:0] Data_in_t [0:cycles];
// /////////////////////////////
// Instantiation of the Design
// /////////////////////////////
stopwatch UUT (.CLK (tbclk), .RESET (tbreset), .STRTSTOP (tbstrtstop),
 .ONESOUT (onesout), .TENSOUT (tensout), .TENTHSOUT (tbtenthsout));
 wire [4:0] tbonesout, tbtensout;
 assign tbtensout = led2hex(tensout);
 assign tbonesout = led2hex(onesout);
///////////////////////////////////////////////////////////////
//EXPECTED RESULTS
///////////////////////////////////////////////////////////////
initial begin
    Data_in_t[1] =10'b1111111110;
    Data_in_t[2] =10'b1111111101;
    Data_in_t[3] =10'b1111111011;
    Data_in_t[4] =10'b1111110111;
```

```
        Data_in_t[5] =10'b1111101111;
        Data_in_t[6] =10'b1111011111;
        Data_in_t[7] =10'b1110111111;
        Data_in_t[8] =10'b1101111111;
        Data_in_t[9] =10'b1011111111;
        Data_in_t[10]=10'b0111111111;
        Data_in_t[11]=10'b1111111110;
        Data_in_t[12]=10'b1111111110;
        Data_in_t[13]=10'b1111111101;
        Data_in_t[14]=10'b1111111011;
        Data_in_t[15]=10'b1111110111;
        Data_in_t[16]=10'b1111101111;
        Data_in_t[17]=10'b1111011111;
        Data_in_t[18]=10'b1110111111;
        Data_in_t[19]=10'b1101111111;
        Data_in_t[20]=10'b1011111111;
        Data_in_t[21]=10'b0111111111;
        Data_in_t[22]=10'b1111111110;
        Data_in_t[23]=10'b1111111110;
        Data_in_t[24]=10'b1111111101;
        Data_in_t[25]=10'b1111111011;
    end
    reg GSR;
    assign glbl.GSR = GSR;
    initial begin
      GSR = 1;
      // ///////////////////////////////
      // Wait till Global Reset Finished
      // ///////////////////////////////
      #100 GSR = 0;
    end

    // ////////////////
    // Create the clock
    // ////////////////
    initial begin
      tbclk = 0;
      // Wait till Global Reset Finished, then cycle clock
      #100 forever #60 tbclk = ~tbclk;
    end
    initial begin
      // //////////////////////////
      // Initialize All Input Ports
      // //////////////////////////
      tbreset = 1;
      tbstrtstop = 1;
      // /////////////////////
      // Apply Design Stimulus
      // /////////////////////
      #240 tbreset = 0;
      tbstrtstop = 0;
      #5000 tbstrtstop = 1;
      #8125 tbstrtstop = 0;
      #500 tbstrtstop = 1;
      #875 tbreset = 1;
      #375 tbreset = 0;
      #700 tbstrtstop = 0;
      #550 tbstrtstop = 1;
      // ///////////////////////////////////////////////////////
      // simulation must be halted inside an initial statement
      // ///////////////////////////////////////////////////////
    //  #100000 $stop;
    end
    integer i,errors;
    //////////////////////////////////////////////////////////////////
    ///////////////
    // Block below compares the expected vs. actual results
    // at every negative clock edge.
    //////////////////////////////////////////////////////////////////
    ///////////////
    always @ (posedge tbclk)
    begin
     if (tbstrtstop)
      begin
      i = 0;
      errors = 0;
      end
     else
     begin
        for (i = 1; i <= cycles; i = i + 1)
        begin
         @(negedge tbclk)
         // check result at negedge
         $display("Time%d ns; TBSTRTSTOP=%b; Reset=%h; Expected
    TenthsOut=%b; Actual TenthsOut=%b", $stime, tbstrtstop, tbreset,
    Data_in_t[i], tbtenthsout);
         if ( tbtenthsout !== Data_in_t[i] )
          begin
          $display(" ------ERROR. A mismatch has occurred-----");
          errors = errors + 1;
          end
        end
     if (errors == 0)
      $display("Simulation finished Successfully.");
     else if (errors > 1)
      $display("%0d ERROR! See log above for details.",errors);
     else
      $display("ERROR! See log above for details.");
    #100 $stop;
     end
    end
    endmodule
```

This simple, self-checking testbench design can be ported to any test case—of course, expected-output values and signal names must be modified for reuse. If a check is not needed at each clock edge, the for-loop can be modified as needed.

If the simulation succeeds, the following information is shown on the terminal screen:


ModelSim SE PLUS 5.5a
File Edit Design View Project Run Compare Macro Options Window Help
1000

```
VSIM 14> run 23000 ns
#
# Beginning Simulation...
# Pattern#      0 time    1950: WE=0; Address=0; Data=00; Expected Q=xx; Actual Q=xx
# Pattern#      1 time    2950: WE=1; Address=0; Data=01; Expected Q=01; Actual Q=01
# Pattern#      2 time    3950: WE=1; Address=1; Data=02; Expected Q=02; Actual Q=02
# Pattern#      3 time    4950: WE=1; Address=2; Data=04; Expected Q=04; Actual Q=04
# Pattern#      4 time    5950: WE=1; Address=3; Data=08; Expected Q=08; Actual Q=08
# Pattern#      5 time    6950: WE=1; Address=4; Data=10; Expected Q=10; Actual Q=10
# Pattern#      6 time    7950: WE=1; Address=5; Data=20; Expected Q=20; Actual Q=20
# Pattern#      7 time    8950: WE=1; Address=6; Data=40; Expected Q=40; Actual Q=40
# Pattern#      8 time    9950: WE=1; Address=7; Data=80; Expected Q=80; Actual Q=80
# Pattern#      9 time   10950: WE=0; Address=0; Data=07; Expected Q=01; Actual Q=01
# Pattern#     10 time   11950: WE=0; Address=1; Data=08; Expected Q=02; Actual Q=02
# Pattern#     11 time   12950: WE=0; Address=2; Data=09; Expected Q=04; Actual Q=04
# Pattern#     12 time   13950: WE=0; Address=3; Data=10; Expected Q=08; Actual Q=08
# Pattern#     13 time   14950: WE=0; Address=4; Data=11; Expected Q=10; Actual Q=10
# Pattern#     14 time   15950: WE=0; Address=5; Data=12; Expected Q=20; Actual Q=20
# Pattern#     15 time   16950: WE=0; Address=6; Data=13; Expected Q=40; Actual Q=40
# Pattern#     16 time   17950: WE=0; Address=7; Data=14; Expected Q=80; Actual Q=80
# Pattern#     17 time   18950: WE=1; Address=0; Data=aa; Expected Q=aa; Actual Q=aa
# Pattern#     18 time   19950: WE=0; Address=0; Data=55; Expected Q=aa; Actual Q=aa
# Pattern#     19 time   20950: WE=1; Address=0; Data=55; Expected Q=55; Actual Q=55
# Pattern#     20 time   21950: WE=0; Address=0; Data=aa; Expected Q=55; Actual Q=55
# Good!  End of Good Simulation.
# ** Note: $finish    : F:/projects/appnotes/testbenches/ver_example/test.v(112)
#    Time: 22950 ns  Iteration: 0  Instance: /test
# 1
# Break at F:/projects/appnotes/testbenches/ver_example/test.v line 112

VSIM 15>

Now: 22,950 ns    Delta: 0                    sim:/test
```

Figure 3: **Verilog Example Verification**

VHDL Example

In VHDL, a vector file contains expected results. The VHDL textio package is used to read data from the vector file, and to display error messages. This testbench instantiates the stopwatch design in VHDL.

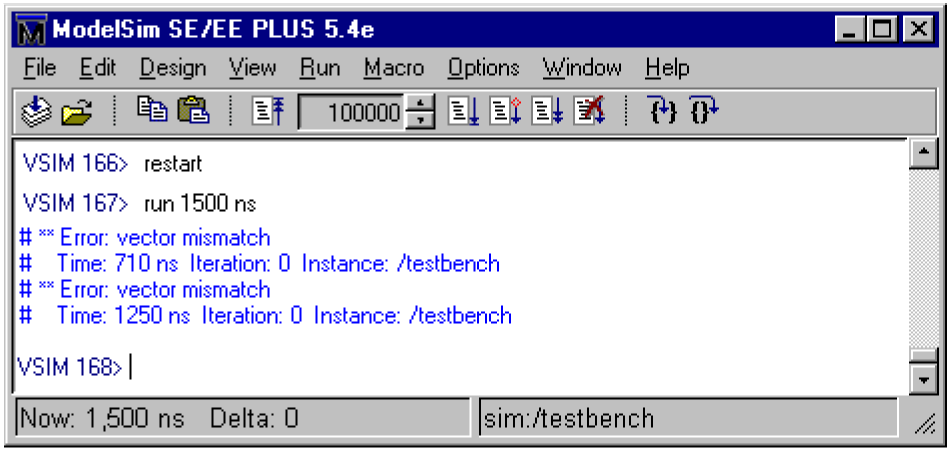
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
LIBRARY ieee;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
ENTITY testbench IS
END testbench;
ARCHITECTURE testbench_arch OF testbench IS
COMPONENT stopwatch
PORT (
  CLK : in STD_LOGIC;
  RESET : in STD_LOGIC;
  STRTSTOP : in STD_LOGIC;
  TENTHSOUT : out STD_LOGIC_VECTOR (9 DOWNTO 0);
  ONESOUT : out STD_LOGIC_VECTOR (6 DOWNTO 0);
  TENSOUT : out STD_LOGIC_VECTOR (6 DOWNTO 0)
);
END COMPONENT;
SIGNAL CLK : STD_LOGIC;
SIGNAL RESET : STD_LOGIC;
SIGNAL STRTSTOP : STD_LOGIC;
SIGNAL TENTHSOUT : STD_LOGIC_VECTOR (9 DOWNTO 0);
SIGNAL ONESOUT : STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL TENSOUT : STD_LOGIC_VECTOR (6 DOWNTO 0);
constant ClockPeriod : Time := 60 ns;
FILE RESULTS: TEXT IS OUT "results.txt";
signal i: std_logic;
BEGIN
UUT : stopwatch
PORT MAP (
  CLK => CLK,
  RESET => RESET,
  STRTSTOP => STRTSTOP,
  TENTHSOUT => TENTHSOUT,
  ONESOUT => ONESOUT,
  TENSOUT => TENSOUT
);
stimulus: PROCESS
begin
reset <= '1';
strtstop <= '1';
wait for 240 ns;
reset <= '0';
strtstop <= '0';
wait for 5000 ns;
strtstop <= '1';
wait for 8125 ns;
strtstop <= '0';
wait for 500 ns;
strtstop <= '1';
wait for 875 ns;
reset <= '1';
wait for 375 ns;
reset <= '0';
wait for 700 ns;
strtstop <= '0';
wait for 550 ns;
strtstop <= '1';
end process stimulus;
clock: process
begin
 clk <= '1';
  wait for 100 ns;
 loop
   wait for (ClockPeriod / 2);
  CLK <= not CLK;
 end loop;
end process clock;
check_results : process
variable tmptenthsout: std_logic_vector(9 downto 0);
variable l: line;
variable good_val, good_number, errordet: boolean;
variable r : real;
variable vector_time: time;
variable space: character;
file vector_file: text is in "values.txt";
begin
  while not endfile(vector_file) loop
   readline(vector_file, l);
   read(l, r, good => good_number);
   next when not good_number;
   vector_time := r * 1 ns;
   if (now < vector_time) then
    wait for vector_time - now;
   end if;
   read(l, space);
   read(l, tmptenthsout, good_val);
   assert good_val REPORT "bad tenthsoutvalue";
   wait for 10 ns;
   if (tmptenthsout /= tenthsout) then
    assert errordet REPORT "vector mismatch";
   end if;
  end loop;
wait;
end process check_results;
end testbench_arch;
library XilinxCoreLib;
CONFIGURATION stopwatch_cfg OF testbench IS
FOR testbench_arch
 FOR ALL : stopwatch use configuration work.cfg_tenths;
 END FOR;
END FOR;
END stopwatch_cfg;
```

The following vector file is used with the testbench above. It contains expected simulation values.

```
-- Vector file containing expected results
0 1111111110
340 1111111110
400 1111111101
460 1111111011
520 1111110111
580 1111101111
640 1111011111
700 1110111111
760 1101111111
820 1011111111
880 0111111111
940 1111111110
1000 1111111110
1060 1111111101
1120 1111111011
1180 1111110111
1240 1111101111
1300 1111011111
1360 1110111111
1420 1101111111
1480 1011111111
```

```
1540 0111111111
1600 1111111110
1660 1111111110
1720 1111111101
1780 1111111011
```

If an error is detected, it is displayed at the simulator prompt. Figure 4 shows errors displayed in the MTI transcript window.



```
VSIM 166> restart

VSIM 167> run 1500 ns
# ** Error: vector mismatch
#    Time: 710 ns  Iteration: 0  Instance: /testbench
# ** Error: vector mismatch
#    Time: 1250 ns  Iteration: 0  Instance: /testbench

VSIM 168> |
```

```
Now: 1,500 ns   Delta: 0        sim:/testbench
```

Figure 4: **Simulator-Prompt Error Report**

Guidelines for Writing Testbenches

This section provides guidelines for writing testbenches. Just as planning a circuit design helps achieve better circuit performance, planning a testbench layout improves simulation verification results.

- *Know the simulator before writing the testbench.*

  Although commonly-used simulation tools conform to HDL industry standards, these standards do not address several important simulation-specific issues. Different simulators have different features, capabilities, and performance characteristics, and produce different simulation results.

  - *Event-based vs. cycle-based simulation*

    Simulators use event-based or cycle-based simulation methods. Event-based simulators schedule a simulator event when an input, signal, or gate changes value. In an event-based simulator, a delay value can be associated with gates and nets to achieve optimum timing simulation. Cycle-based simulators target synchronous designs. They optimize combinatorial logic and analyze results at clock cycles. This feature makes cycle-based simulators faster and more memory efficient than event-based simulators. However, since cycle-based simulators do not allow detailed timing specificity, they are not as accurate. For further information on these differences, see Digital Logic Simulation: Event-Driven, Cycle-Based, and Home-Brewed, available at **http://www.ednmag.com/ednmag/reg/1996/070496/14df4.htm**.

  - *Scheduling events*

    Event-based simulator vendors employ different algorithms for scheduling simulation events. Therefore, events that occur at the same simulation time may be scheduled in a different sequence (with delta delays inserted between each event) depending on the scheduling algorithm used by the simulator. To avoid algorithmic dependencies and assure correct results, an event-driven testbench should specify an explicit stimulus sequence.

- *Avoid using infinite loops*

  When an event is added to an event-based simulator, CPU and memory usage increases, and simulation processing slows. Unless critical to the testbench, infinite loops should not be used to provide design stimulus. Typically, clocks are specified inside an infinite loop (for example, "forever" loops in Verilog), but not other signal events.

- *Break up stimuli into logical blocks*

  Within a testbench, all initial (Verilog) and process (VHDL) blocks run concurrently. The testbench stimulus sequence becomes easier to implement and review if unrelated stimuli are divided into separate blocks. Since each concurrent block runs relative to simulation time zero, passing stimulus is easier with separate blocks. The use of separate stimulus blocks results in testbenches that are easier to create, maintain, and upgrade (see Advanced Testbench Techniques, below, for examples of this technique).

- *Avoid displaying unimportant data*

  Testbenches for large designs may contain in excess of 100,000 events and numerous signals. Displaying a large amount of simulation data slows simulation considerably. It is best to sample only relevant signals every “n” clock cycles to assure adequate simulation speed.

Xilinx Simulation Flow Tips

Configuration Statement (VHDL)

A VHDL configuration statement allows an entity to be linked to a specific architecture for synthesis or simulation. In a Xilinx CORE Generator VHDL functional simulation flow, a configuration statement is used to bind the core simulation model to the design. If the core simulation model is not bound to a design, simulation will not work properly. For an example of configuration statement use, see the VDHL code example in Self-Checking Testbenches, above. Detailed information on the use of configuration statements in Xilinx CORE Generator designs can be found in the Modelsim VHDL simulation tutorial, available at http://support.xilinx.com/support/techsup/tutorials/tutorials31i.htm#Modelsim.

Initializing Block RAMs for Simulation

By default, Xilinx Virtex™ block RAMs are initialized to zero in all data locations, starting at time zero. For a post-NGDBuild, post-MAP, or Post-PAR (timing) simulation, block RAMs initialize to values specified in the user constraints file (UCF) or through INIT attributes specified in the input design file to NGDBuild. For a pre-synthesis or post-synthesis (pre-NGDBuild) functional simulation, a configuration statement must be used to supply initial values to block RAM. The following is an example a configuration statement used to initialize block RAM.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.vcomponents.all;
configuration cfg_ex_blkram_tb of ex_blkram_tb is
 for tb
  for uut : ex_blkram use entity work.ex_blkram(struct);
   for struct
    for INST_RAMB4_S4 : RAMB4_S4 use entity
     unisim.RAMB4_S4(RAMB4_S4_V)
     generic map (INIT_00 =>
X"1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100
",
INIT_01 =>
X"3F3E3D3C3B3A393837363534333231302F2E2D2C2B2A29282726252423222120
",
.
.
.
INIT_0F=>
X"FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1F0EFEEEDECEBEAE9E8E7E6E5E4E3E2E1E0
");
    end for;
   end for;
  end for;
 end for;
end cfg_ex_blkram_tb;
```

Advanced Testbench Techniques

Breaking Up Stimulus Blocks with Tasks and Procedures

When creating larger testbenches, stimuli should be partitioned to aid code clarity and facilitate modifications. Tasks (Verilog) or procedures (VHDL) can be used to partition signals. In the following example, the testbench stimulates a SDRAM controller design. The design includes blocks of repetitive stimulus, so the testbench partitions the stimulus by declaring separate tasks which are called later in the testbench to exercise separate design functionalities.

Verilog Example:

```
task addr_wr;
   input [31 : 0] address;
   begin
 data_addr_n = 0;
 we_rn = 1;
 ad = address;
   end
 endtask
task data_wr;
   input [31 : 0] data_in;
   begin
 data_addr_n = 1;
 we_rn = 1;
 ad = data_in;
   end
 endtask

task addr_rd;
   input [31 : 0] address;
   begin
 data_addr_n = 0;
 we_rn = 0;
 ad = address;
   end
 endtask

task data_rd;
   input [31 : 0] data_in;
   begin
 data_addr_n = 1;
 we_rn = 0;
 ad = data_in;
   end
 endtask
```

```
task nop;
   begin
 data_addr_n = 1;
 we_rn = 0;
 ad = hi_z;
   end
 endtask
```

These tasks specify separate elements of design functionality-address read and write, data read and write, or nop (no operation). Once specified, these tasks can be called within stimulus processes as follows:

```
Initial begin
nop ;                    // Nop
#( 86* `CYCLE +1); addr_wr (32'h20340400);  // Precharge, load
Controller MR
#(`CYCLE); data_wr (32'h0704a076);   // value for Controller MR
#(`CYCLE); nop ;              // Nop
#(5 * `CYCLE); addr_wr (32'h38000000);   // Auto Refresh
#(`CYCLE); data_wr (32'h00000000);   //
#(`CYCLE); nop ;               // Nop
…
…
end
```

VHDL Example:

Below is a VHDL testbench for the same design, broken up into separate procedures:

```
Stimulus : process
 procedure addr_wr (address: in std_logic_vector(31 downto 0)) is
 begin
  data_addr_n <= ‘0’;
  we_rn <= ‘1’;
  ad <= address;
 end addr_wr;
 procedure data_wr (data_in: in std_logic_vector(31 downto 0 )) is
 begin
  data_addr_n <= ‘1’;
  we_rn <= ‘1’;
  ad <= data_in;
 end data_wr;

 procedure addr_rd (address: in std_logic_vector(31 downto 0)) is
 begin
  data_addr_n <= ‘0’;
  we_rn <= ‘0’;
  ad <= address;
 end addr_rd;

 procedure data_rd (data_in: in std_logic_vector(31 downto 0)) is
 begin
  data_addr_n <= ‘1’;
  we_rn <= ‘0’;
  ad <= data_in;
 end data_rd;

 procedure nop is
 begin
  data_addr_n <= ‘1’;
```

```
   we_rn = '0';
   ad = 'Z';
  end nop;
 begin
  nop ;     -- Nop
   wait for 200 ns;
  addr_wr (16#20340400#);  -- Precharge, load Controller MR
   wait for 8 ns;
  data_wr (16#0704a076#);   -- value for Controller MR
   wait for 8 ns;
  nop ;               -- Nop
   wait for 40 ns;
  addr_wr (16#38000000#);  -- Auto Refresh
   wait for 8 ns;
  data_wr (16#00000000#);
   wait for 8 ns;
  nop ;               -- Nop
 ..
 ..
```

Breaking up stimulus into separate tasks makes stimulus passing easier to implement, and makes the code more readable.

Controlling Bidirectional Signals in Simulation

Most designs use bidirectional signals, which must be treated differently than unidirectional signals in a testbench.

VHDL Example:

The following is a VHDL bi-directional signal example:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity bidir_infer is
port (DATA : inout STD_LOGIC_VECTOR(1 downto 0);
READ_WRITE : in STD_LOGIC);
end bidir_infer;
architecture XILINX of bidir_infer is
signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
begin
process(READ_WRITE, DATA)
begin
 if (READ_WRITE = '1') then
  LATCH_OUT <= DATA;
 end if;
end process;
process(READ_WRITE, LATCH_OUT)
begin
 if (READ_WRITE = '0') then
  DATA(0) <= LATCH_OUT(0) and LATCH_OUT(1);
  DATA(1) <= LATCH_OUT(0) or LATCH_OUT(1);
 else
  DATA(0) <= 'Z';
  DATA(1) <= 'Z';
 end if;
end process;
end XILINX;
```

To access the bidirectional DATA signal in the above example, a testbench can be setup as follows:

```
library ieee;
use ieee.std_logic_1164.all;
Entity testbench is
End testbench;
Architecture test_bidir of testbench is
Component bidir_infer
 port (DATA : inout STD_LOGIC_VECTOR(1 downto 0);
    READ_WRITE : in STD_LOGIC);
end component;
signal read_writet: std_logic;
signal datat, data_top : std_logic_vector(1 downto 0);
begin
datat <= data_top when (Read_writet = '1') else (others => 'Z');
data_top <= datat when (Read_writet = '0') else (others => 'Z');
uut : bidir_infer port map (datat, read_writet);
process begin
 read_writet <= '1';
 data_top <= "10";
  wait for 50 ns;
 read_writet <= '0';
  wait;
 end process;
end test_bidir;
```

The bidirectional bus is controlled from the testbench, and the value of the bidirectional bus is accessed through the data_top signal.

Verilog Example:

The following is an example of an inferred bidirectional bus in a Verilog design:

```
module bidir_infer (DATA, READ_WRITE);
input READ_WRITE ;
inout [1:0] DATA ;
reg [1:0] LATCH_OUT ;
always @ (READ_WRITE or DATA)
begin
 if (READ_WRITE == 1)
  LATCH_OUT <= DATA;
end
assign DATA = (READ_WRITE == 1) ? 2'bZ : LATCH_OUT;
endmodule
```

The Verilog testbench can be set up as follows:

```
module test_bidir_ver;
reg read_writet;
reg [1:0] data_in;
wire [1:0] datat, data_out;
bidir_infer uut (datat, read_writet);
assign datat = (read_writet == 1) ? data_in : 2'bZ;
assign data_out = (read_writet == 0) ? datat : 2'bZ;
initial begin
 read_writet = 1;
 data_in = 11;
 #50 read_writet = 0;
end
endmodule
```

In the above testbench, the data_in signal supplies the stimulus to the bi-directional DATA signal in the design, and the data_out signal reads back the value of the DATA signal.

Initializing Memory for Simulation

Please refer to the Xilinx Simulation Flow Tips, above.

Useful Language Constructs

Verilog

Useful Verilog language constructs, such as $monitor, $display, and $time, are discussed in the Verilog testbench examples, above. This section discusses additional Verilog constructs that can be used in a testbench.

force/release

The force and release statements can be used to override procedural assignments made to a register or net. These constructs are commonly used to force specific design behavior. Once a forced value is released, the signal retains its state until a new value is passed through a procedural assignment. The following is an example of force and release statement usage:

```
module testbench;
..
..
initial begin
reset = 1;
force DataOut = 101;
#25 reset = 0;
#25 release DataOut;
..
..
end
endmodule
```

assign/deassign

The assign and deassign statements are similar to the force and release statements, but assign and deassign apply only to registers in a design. They are normally used to set input values. Like a force statement, the assign statement overrides values passed by procedural statements. The following is an example of assign and deassign statement usage:

```
module testbench;
..
..
initial begin
reset = 1;
DataOut = 101;
#25 reset = 0;
release DataOut;
..
..
end
initial begin
#20 assign reset = 1;// this assign statement overrides the earlier
statement #25 reset = 0;
#50 release reset;
endmodule
```

timescales

The timescale directive is used to specify the unit time step for the testbench. It also affects simulator precision. The syntax for this directive is:

`timescale *reference_time/precision*

Reference_time is the unit time for measurement. Precision determines the precision to which the delays are rounded-off, and sets the unit time step for simulation. The following is an example of `timescale usage:

```
`timescale 1 ns / 1 ps
// this sets the reference time to 1 ns and precision to 1 ps.
module testbench;
..
..
initial begin
#5 reset = 1; // 5 unit time steps correspond to 5 * 1ns = 5ns in
simulation time
#10 reset = 0;
..
end
initial begin
$display (“%d , Reset = %b”, $time, reset); // this display
// statement will get executed
// on every simulator step, ie, 1 ps.
end
endmodule
```

If the simulation uses timing-delay values, simulation must run at a precision greater than the smallest delay (in order to incorporate the delay). For example, if a 9 ps delay is used in simulation libraries, the precision of the simulation must be 1 ps to accommodate the 9 ps delay.

Reading Memory Initialization Files

Verilog provides the $readmemb and $readmemh commands to read an ASCII file to initialize memory contents. This command can be used to initialize Xilinx BlockRAM or SelectRAM components in simulation. The syntax is as follows:

```
$readmemb (“<design.mif>”, design_instance);
```

The MIF is the Memory Initialization File created by coregen. The user specifies MIF contents.

VHDL

In addition to the VHDL commands discussed in previous sections (assert, wait, and report), the following constructs are available to aid VHDL testbench creation:

meminitfile

VHDL provides a meminitfile record that can be used to input the contents of a memory module. The following is the syntax for this construct:

```
FILE meminitfile: TEXT IS IN “<design.mif>”;
```

The MIF is the Memory Initialization File created by coregen. The user specifies MIF contents.

Coding Style Guidelines

The following coding guidelines help create code that is easier to understand and maintain:

Indentation

Always indent code to make it more readable. An indentation width of three or four spaces is recommended. A indentation width of five or more spaces often leaves little space at the right margin, while a indentation width of less than three spaces results in too small an indent.

File Naming

Always maintain the ".v" (Verilog) or ".vhd" (VHDL) filename extension in source filenames. If these standard extensions are changed, some editors and filters will be unable to recognize the source files.

Signal Naming

Use the same case—lower case is recommended—for all user signals. Verilog is case-sensitive, and misplaced capitalization can cause a design to fail synthesis and simulation. Also, use of a consistent signal name format style makes signal names easier to locate in a source file. Use short, descriptive signal names. Short names are easier to type, and descriptive names help document signal functionality.

Commenting

Comment testbench code liberally. Comments are invaluable to others who inherit and reuse code. Besides, Verilog and VHDL code constructs are cryptic—commented code fills in important details that greatly increase source code clarity and reusability.

Design Structure

Keep one physical file for each module or entity. Separate files for separate modules and entities makes the design easier to maintain.

For additional information, consult HDL reference books. Many contain comprehensive coding guidelines.

Refer to the FPGA Design Reuse Guide, available on the web at:

http://www.xilinx.com/ipcenter/designreuse/xrfg.htm

Conclusion

Testbenches provide engineers with a portable, upgradable verification flow. With the availability of mixed-language simulators, designers are free to use their HDL language of choice to verify both VHDL and Verilog designs. High-level behavioral languages facilitate the creation of testbenches that use simple constructs and require a minimum amount of source code. Designs benefit from self-checking testbenches that automate the verification of correct design results during simulation.

Xilinx Foundation™ ISE v3.1i is designed to provide a smooth, integrated HDL design flow. Synplicity's Synplify, Synopsys FPGA Express, and Xilinx Synthesis Technology (XST), along with Xilinx Foundation software, work extremely well together to synthesize code. Foundation ISE can be integrated with Modelsim (XE, PE, or SE) to simulate the design, Xilinx HDL Bencher to automate testbench creation, and Xilinx StateCad to create state-machine code. For complete information on Foundation ISE and its suite of integrated components, go to **http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp**.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/11/01	1.0	Initial Xilinx release.
05/17/10	1.1	Fixed typo on page 2.