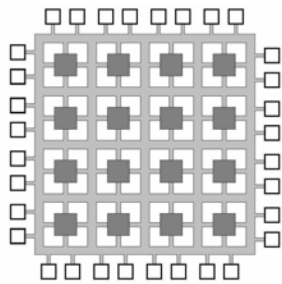

Chip-Level FPGA Verification

How To Use A VHDL Test Bench To Perform A System Auto Test

By Jeffery A. Smith, Senior Development Engineer, Rochester MicroSystems, Inc.

Synopsis

This paper explores the advantages of chip-level FPGA verification, and describes a basic procedure for developing a system auto test with a VHDL virtual test bench. It is intended for engineers and technicians with a good understanding of VHDL programming and test bench operation.

This approach takes FPGA verification beyond the basic level of module-level simulation and waveform inspection. While it is not appropriate for small, glue logic FPGA verification or full ASIC development, it is an ideal strategy for moderate-to-large FPGA applications where maximum code coverage and quick time-to-market are the priorities.

As shown in Figure 1, the architecture for this approach is a VHDL test bench which utilizes a complete range of test cases, bus-functional models and auto-testing monitors to stimulate and evaluate all aspects of FPGA performance.

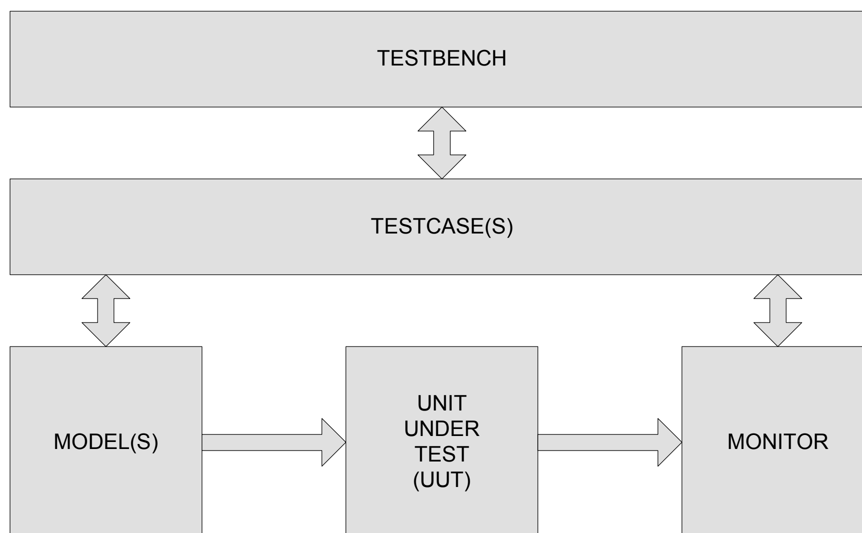

Figure 1: The VHDL test bench controls and selects the test cases; the test cases define the bus-functional models which stimulate the UUT and the auto-testing monitor which tests the UUT output.

Chip-Level FPGA Verification

How To Use A VHDL Test Bench To Perform A System Auto Test

By Jeffery A. Smith, Senior Development Engineer, Rochester MicroSystems, Inc.

CONTENTS

Synopsis

1. Introduction: The Need For Chip Level Verification

2. Initial Considerations
   - Software Issues
   - Writing Reusable Code
     - • Naming Conventions
     - • Defining Parameters
     - • Establishing Subprograms
   - Scheduling Code Reviews

3. Architecture For Chip Level Verification
   - The VHDL Test Bench
     - • Control
     - • Report
     - • File I/O
     - • Random Numbers
   - Test Cases
   - Bus-Functional Models
   - Self-Checking Monitors

4. Additional Considerations
   - Asynchronous Input Timing
   - Code Coverage Tools
   - Gate Timing Analysis Tools

5. Summary

6. References

1. Introduction: The Need For Chip-Level FPGA Verification

FPGA devices now rival ASICs in their complexity; teams of engineers are needed to design and develop million-gate FPGA devices with high pin-count packages. These fine-grid components are difficult to fully debug in the laboratory while time-to-market considerations often restrict board test and system verification efforts.

How can we verify the functionality of complex FPGA devices and still maintain their time-to-market advantage? For this level of complexity, a new strategy for code verification is required; it is no longer adequate to merely simulate at the module level and inspect the waveforms. A properly-executed, chip-level verification strategy will provide maximum code coverage within a minimum amount of time. The goal of the strategy presented below is to provide a simple but thorough chip-level verification process which does not conflict with modern time-to-market requirements.

The most common FPGA verification problem is often lack of planning. If verification is an afterthought or a quick test bench simulation, significant design problems may remain undiscovered until board or system test. The longer problems go undetected, the more time is generally required to resolve them; efforts to recreate and debug a design issue in the lab can result in significant costs and launch delays. Even worse is the discovery of a latent design issue in the field.

It is obviously best to catch these problems during the development stage with a concentrated verification effort. A reasonable benchmark is that the verification effort should be equivalent in time to the design effort. This may be difficult to initially justify, but the delays caused by having to resolve design issues in the lab or field, as well as fine-grid signal accessibility limitations, make it a cost-effective target. The extra benefit of a VHDL test bench written at the chip level is that it can be used for verification at multiple stages: behavioral, post synthesis, and gate simulation. A VHDL auto test bench is also easier to control, maintain and reuse, and ultimately offers the best compromise between absolute design integrity and practical time-to-market considerations.

2. Initial Considerations

Software & Documentation

As software, VHDL benefits from the same controls as C++ or any other programming language. Maintainability, repeatability and reusability are all important factors in achieving time-to-market goals. Source control is essential for releases; there must be an accurate, documented history of the changes and updates to each module. To organize your source control efforts, consider purchased programs such as PVCS or freeware such as CVS.

Bug tracking can often be accomplished with a simple spreadsheet. It is important to document each instance, assign team members to investigate, and record the final resolution so that nothing is missed or forgotten.

A reader-friendly coding style will make it easier for team members to implement the functionality. The use of white space, comment line separators and a comment header for procedures will make your code more readable and understandable.

Writing Reusable Code

Writing standardized, reusable code is an effective way to reduce the overall cost and time of verification efforts. The utilization of naming conventions, subprograms, system parameters, and unconstrained parameters will maximize the flexibility of your VHDL code.

• Naming Conventions

If VHDL code is to be maintained or reused by other engineers it is much more efficient to use common naming conventions such as those shown in Figure 2. Naming conventions reduce the risk of ambiguity and simplify the editing process.

Suffix	Description
_clk	All clocks
_rst	All resets
_l	Indicates the signal is active low
_r	Indicates the signal has been registered
_r2	Indicates the signal has been registered twice
_asyn	Indicates asynchronous signal

Figure 2: The use of typical naming conventions reduces ambiguity and simplifies editing.

• Parameters

VHDL parameters are implemented by defining constants. Local constants can be defined in the architecture, while global constants are best defined in a separate package for reuse. As shown in the example below, a FIFO model original which is 16 bit x 256 can easily be changed to 32 bit x 128 with careful definition of local and global constants.

Constant	Code
system clock period	constant CLK_PERIOD : time := 50 ns
FIFO bus size	constant ADDR_SIZE : integer := 32-1
FIFO bus size	constant DATA_SIZE : integer := 16-1
FIFO depth	constantFIFO_DEPTH : integer := 256-1

Figure 3: VHDL parameters are implemented with local and global constants.

Unconstrained parameters are often used to make the test bench more flexible. For example, the packet size may vary with different test cases. By using an unconstrained parameter, the size of the data can actually be determined by the data itself. As shown in Figure 4, unconstrained parameters are constructed with the definition “NATURAL” and attributes such as “range”, “left”, “right”, etc.

Unconstrained Parameter
TYPE frame_array IS ARRAY (natural RANGE <>) OF integer; for i in frame_array'range loop …

Figure 4: An example of an unconstrained parameter.

• Subprograms

Abstracting functionality into subprograms whenever possible is good software practice. For example, a CPU Read can be placed in a procedure subprogram to make it more maintainable and reusable. It is important to keep in mind that VHDL procedures are sequential when called in a process and therefore have no view of previous events. Since the parameters are local, a procedure cannot check for setup stability on an input signal. So, setup stability needs to be verified before the procedure call. Procedures can check for stability hold with internal signals. VHDL supports procedure overloading. Utilize overloading to define procedures with several parameters list options.

Two examples of VHDL subprograms are shown below. The first procedure is for setting up a clock delay; the code in the second example initializes or asserts reset.

```
-------------------------------------------------------------------------------
procedure clk_dly(clk_cnt:in integer; signal clk:in std_logic) is
begin

   for I in 1 to clk_cnt loop
      wait until rising_edge(clk);
   end loop;
end clk_dly;
-------------------------------------------------------------------------------
procedure init_reset(signal RESET_IN_L_Signal: inout std_logic) is
begin

   RESET_IN_L_Signal <=  '0'; -- assert
   wait for 126 ns;   -- Initialize reset
   RESET_IN_L_Signal <=  '1'; -- negate
   wait for 100 ns;    -- let reset trickle through
end init_reset;
```

Figure 5: examples of VHDL subprograms

Scheduling Code Reviews

Code reviews are important aspect of any chip-level verification effort. Regularly scheduled reviews -- with two to four engineers reviewing code, syntax, logic and system functionality -- will almost always turn up mistakes or programming which can be improved. It is more cost-effective to find and repair mistakes at the early stages of the development process; an hour-long review with four engineers will quickly pay for itself if only one error is found and corrected.

Objectivity and fresh perspective are prized assets for a code review team. To make your reviews more productive, it is good practice to include engineers or outside design partners who had no role in writing the original code.

3. Architecture For Chip-Level Verification

The basic architectural concept for chip-level verification is straightforward -- stimulate all inputs and test all outputs. System input signals such as clock and reset are generated in the test bench architecture. External devices such as CPU or FIFO can be emulated by employing reusable, bus-functional models. All FPGA outputs are captured and self-tested by monitors. Test cases for different system scenarios and data options such as nominal, minimum, maximum, and error cases drive the simulation.

The FPGA design will often determine the best approach to test bench configuration. With a "black box" approach, the test bench has no interaction of any kind with the FPGA design; a "grey box" approach allows the test bench to exert some control. For some designs, it may be desirable to make the test bench capable of writing a control register. For example, a long simulation time would be required if the design has a ten-second watchdog. By giving the test bench some internal control, the simulation time can be reduced.

The VHDL Test Bench

The VHDL test bench is the top level module that selects the appropriate stimulus signal for a specific system scenario, and interconnects the unit under test (UUT) with the stimulus, the models and monitors. It is beneficial to have a single test bench that selects and runs a variety of test cases. A single test bench is more maintainable; for example, if a pin is changed, a single edit will keep all test cases functioning. The test bench should be designed with the ability to select which tests to run and stop automatically, and to report test results. A test bench may have file input or output.

It requires careful planning to define the timing of the stimulus and response signals in the top level architecture. Signals that run in the main process are sequential. Signals that run outside the main process are concurrent. Examples of typical concurrent signals and processes are shown below.

```
 constant CLK_PERIOD : time   := 50 ns;
 signal tb_system_clk : std_logic := 0;

 ----------------------------------------------------------------------
-- concurrent continuous signal
tb_system_clk <= not tb_system_clk after CLK_PERIOD/2;

----------------------------------------------------------------------
-- concurrent stimulus
video_stim:process -- infinite video loop, assert enable & go
   variable count:integer:=1;
  begin
      wait until (video_en='1');
      while (video_en = '1') loop
      send_video(i,count,pixels,lines,video_in, video_tb);
      count := count + 1;
      end loop;
  end process video_stim;
----------------------------------------------------------------------
-- concurrent monitor example
video_monitor : process
begin
    wait until eof = '1';
    if video_en = '1' then
       capture_video(video_tb);
    end if;
    wait;
```

```
end process video_monitor;
```

Figure 6: examples of "concurrent" definitions

• Control

The selection of test cases can be accomplished in several ways. The selection information can be read from an external file, from defined parameters within the test bench architecture, or from defined parameters in an external package. The external package approach works well because is compiles quickly and isolates any editing steps from the test bench itself. The example below shows how to define parameters for selecting which test case should be run. Each test case is specified with a control if/then statement, so that the parameters can be defined as true or false.

```
-- define true for test(s) to be run (true or false )
constant do_all: boolean              := false;
constant do_video_min: boolean    := true;
constant do_video_max: boolean   := false;

   if do_video_min or do_all then…

   if do_video_max or do_all then…
```

Figure 7: how to define parameters for selecting test cases

The test bench should be designed to run for the full duration of all the selected test cases and then stop automatically. Since the overall running time is dynamic, running a test bench for a fixed time period may provide inaccurate results and should be avoided. The VHDL code for completing all test cases and automatic stop is shown in Figure 8.

```
assert false report "**** SIMULATION COMPLETE ****"
severity Failure; -- stop sim
wait; -- NEED a wait at the end of a testbench process
```

Figure 8: how to set the test bench for auto stop after all test cases have been run

• Report

The test bench should be designed to report the results of all test cases with run time status reports. It is good practice to have the test bench report each test failure and display the relevant details next to the expected results, along with a time stamp. To provide finer resolution, it is also helpful get a report of the specific subtest within the test case in question. A constant can be added which will turn these reports on or off, similar to C software debugging with printf statements. Reports can be sent to display, file or both. The example below shows how to set up a video test case status report:

```
if v_verbose then
        assert false report "**** TEST Video In Minimum  ****"
        severity note; --   (severity note; error;  failure)
    end if;
```

Figure 8: how to set the test bench for auto stop after all test cases have been run

• File I/O

Setting up file I/O for VHDL can be a cumbersome process. Each line must be generated before it can actually be written; the parameters of each line must be parsed after the line is read. Fortunately, the process can be simplified by employing a utility subprogram that can be reused for other projects. The subprogram will be easier to maintain if the file name is defined in the entity or package.

The sample file I/O utility subprogram shown in Figure 9 includes parameter settings, read_test and write_string.

```
file tb_control_file : text is in  "tb_control.txt";
file tb_report_file : text is out "tb_report.txt";
 -----------------------------------------------------------------
  procedure read_test(which_test:out integer) is
     file file_in: text is in TB_FILE_IN;
     variable line_in : line;
     variable err: bit;
begin   -- read the tests to perform from control file
      readline(tb_control_file, line_in);
      read(line_in, which_test);
     wait for 10 ns;
     assert z = err report "z incorrect" severity error;
     wait;
end;
-----------------------------------------------------------------
procedure wr_string(string_out:in string)is
    variable line_out  : line;
begin
    write(line_out, string_out);
    writeline(OUTPUT, line_out); -- output display

    write(line_out, string_out);
    writeline(tb_report_file,line_out); -- output file
end wr_string;
```

Figure 9: A utility subprogram simplifies File I/O setup.

• Random Numbers

Use a random number generator to provide more realistic test bench model parameters. For example, a propagation delay for a CPU bus-functional model can be set to vary according to a random number to provide a more realistic simulation. VHDL does not have a built-in random function, but there are several packages readily available.

```
cpuOut.oe_n <= '0';
cpuOut.addr <= conv_std_logic_vector(addr,ADDR_SIZE+1);
wait for tsh * rnd_pkg.random;
cpuOut.re_n <= '0' after (tdRd) ;
```

Figure 10: Use a random number generator to provide a more realistic simulation.

Bus-Functional Models

A bus-functional model (BFM) emulates the timing and function of the bus. For example, a CPU BFM emulates the bus signals, ce_n, data and address. A BFM does not emulate the processor boot, fetch, or run mode. A BFM simply emulates the IO signal's timing. When implementing a BFM and utilizing procedure calls, each bus interface signal needs to be passed from the test bench to the model with each procedure call. The length of the parameterize list can be minimized by creating a record for the bus signals, as shown below in Figure 11. If this type of record is utilized, then each interface signal only needs only be passed once.

```
--------------------------------------------------------------------------------
constant tsRd      : time := 1.0 ns; -- time read setup
constant thRd      : time := 1.0 ns; -- time read hold
constant tdRd      : time := 1.0 ns; -- time read data
--------------------------------------------------------------------------------
TYPE cpuOut_type IS RECORD
  cs_n  : std_logic
  addr  : std_logic_vector(ADDR_SIZE downto 0);
--------------------------------------------------------------------------------
procedure cpuread (addr:in integer; rdata:out integer;
                   signal cpuIn  :in cpuIn_type;
                   signal cpuOut :out cpuOut_type ) is
begin
        wait until rising_edge(cpuIn.clk);
        wait for tsRd;
        cpuOut.cs_n <= '0';
        cpuOut.addr <= conv_std_logic_vector(addr,ADDR_SIZE+1);
        clk_dly(wait_st,cpuIn.clk);
        rdata := conv_integer(cpuIn.data); -- read data
        cpuOut. cs_n <= '1' after (tdRd);
        wait for (thRd);
        cpuOut.cs_n <= '1';
        cpuOut.addr <= (others => '0');
end cpuread;
```

Figure 11: Creating a record for bus signals helps to minimize the parameterize list.

Self-Checking Monitors

The monitors capture and test all output signals from the FPGA. Defining monitors and test vectors can be challenging. Normally, the first step is to define the monitors as concurrent or sequential. It is advantageous to minimize the number of processes which run concurrently; if a monitored signal can be sequentially called by a subprogram, simulation run times can be reduced. For example, a subprogram could be used to monitor and test data when a test case sends a video frame. However, if the output can occur at anytime, then the monitor should be defined as concurrent in the top level test bench architecture.

Test vectors and their sources should be viewed from the system level. Test vectors are the stimulus data and can be defined as files, packages, or processes depending on the test bench requirements. If there are a large number of vector types, the best solution might be to define a process to generate them. If a test vector is a large, a bitmap file may be used. Finally, it may be best to define the test vectors in a package if they are limited in size and type. It is also good practice to use proven test vectors (sometimes called "golden vectors") to test the captured data.

Monitor timing requirements are defined by the external requirements of the system. If the data is being sent to a FIFO, refer to the FIFO datasheet for timing requirements. Use the attribute “stable” to verify setup and hold.

```
---------------------------------------------------------------------
-- subprogram example of monitor
procedure test_data(data_in:in frame_type; data_tb:in frame_type)is
begin
   if data_in /= data_tb then
      log_error (" VinData", data_in, data_tb);
      assert false
      severity error;
  end if;
---------------------------------------------------------------------
-- concurrent monitor example located in testbench architecture
capture_video : process
begin
   wait until eof = '1';
   if video_test_en then
      capture_video(video_clk, video_in(x,y));
   wait;
end process capture_video;
```

Figure 12: Examples of test monitors

Test Cases

The UUT needs to be exercised with a variety of test cases which represent a broad range of potential system scenarios. To cover as many scenarios as possible, involve your entire team in the process. The test cases should be defined to reflect the system output (frames of video, packets of data, etc.) and should be modeled for nominal, minimum, maximum and error cases. Error cases should include scenarios such as wrong frame size, corrupt header, wrong number of pixels, etc. For example, a test case for the fiber optic protocol FICON would model the smallest packet size, largest packet size, a packet with a corrupted header, etc. A NTSC video test case could be set up for too few lines, too many pixels or a corrupted trailer.

The final test case should be set up to “test the test”. Insert a real error and verify that the self-testing monitors catch the problem. This can be accomplished by inserting an invalid intermediate test bench signal.

Although test cases can be written in the test bench architecture, it is better practice to use a subprogram if they are very complex or if there are a large number of cases. Figure 13 provides an example of a frame minimum test case.

```
   if do_video_min or do_all then
      if v_verbose then
               assert false report "**** Frame minimum Test Case ****"
               severity note;
      end if;

      test_case <= 16#100#;  -- used for tracking
      init_reset(reset_tb_n); -- reset everyone
```

```
    send_packet(packet_data, min, fail_none);
    wait_clocks (100,system_tb_clk);
    test_packet(packet_data, packet_tb);
end if;-- do_video_min
```

Figure 13: A frame minimum test case.

4. Additional Considerations

Asynchronous Timing Verification

Asynchronous inputs cannot be accurately modeled for timing; the time must be verified empirically. While a 50 MHz system has a period of 20 ns, the simulation tool typically has a resolution of only 1ps. This dictates 20,000 timing possibilities on the input. It would not be practical or realistic to model and simulate each of these possibilities. Although beyond the scope of this paper, a more efficient approach would be to verify that time of flight is sufficient by performing a timing analysis.

Code Coverage Tools

Code coverage is provided as an additional feature by most simulation tools. This useful tool will provide a report on the overall percentage of code covered by the simulation run, an excellent measure of the simulation's effectiveness. Code coverage data is useful to have when delivering code to a customer or another group.

Gate Timing Analysis Tools

Gate Timing Analysis verifies the setup, hold, stability, and skew of the design. Manufacturers of FPGAs typically supply a tool to verify gate timing: "Static Timing Analysis" for Xilinx Foundation; "Timing" for Actel Designer; "Analyzer" for Aletra Quartus II. Refer to the FPGA manufacturer for more information.

5. Summary

As FPGA components become more complex and time-to-market considerations become more critical, a new strategy for FPGA code verification is required. A properly-executed, chip level verification strategy will provide maximum code coverage in a minimum amount of time. A VHDL test bench written at chip level is easy to maintain and reuse, and can also be utilized for verification at the behavioral, post synthesis and gate simulation levels.

With careful consideration of the test bench architecture and a fully developed array of test cases, a VHDL auto test bench offers the best compromise between absolute design integrity of complex FPGA devices and practical time-to-market considerations.

6. References

"Writing Test Benches: Function Verification of HDL Models" by Janick Bergeron
"Reuse Methodology Manual" by Michael Keating and Pierre Bricaud
"VHDL for Logic Synthesis" by Andrew Rushton
"HDL Chip Design" by Douglas J. Smith