Verification Strategies

Author: Rudolf Usselmann
rudi@asics.ws

Rev. 0.1
February 4, 2001

Preliminary Draft

Revision History

Rev.	Date	Author	Description
0.1	4/2/01	Rudolf Usselmann	First Draft

1

Introduction

This document describes the verification procedures and strategy for individual IP Cores and for the entire integrated circuits.

To successfully verify a Device Under Test (DUT) it is essential to have a verification strategy and verification plan before starting to write a test bench. This document will outline the overall strategy for verification. Each DUT, will have a unique verification plan associated with it, addressing the capabilities and challenges for verification.

(This page intentionally left blank)

2

General Guidelines

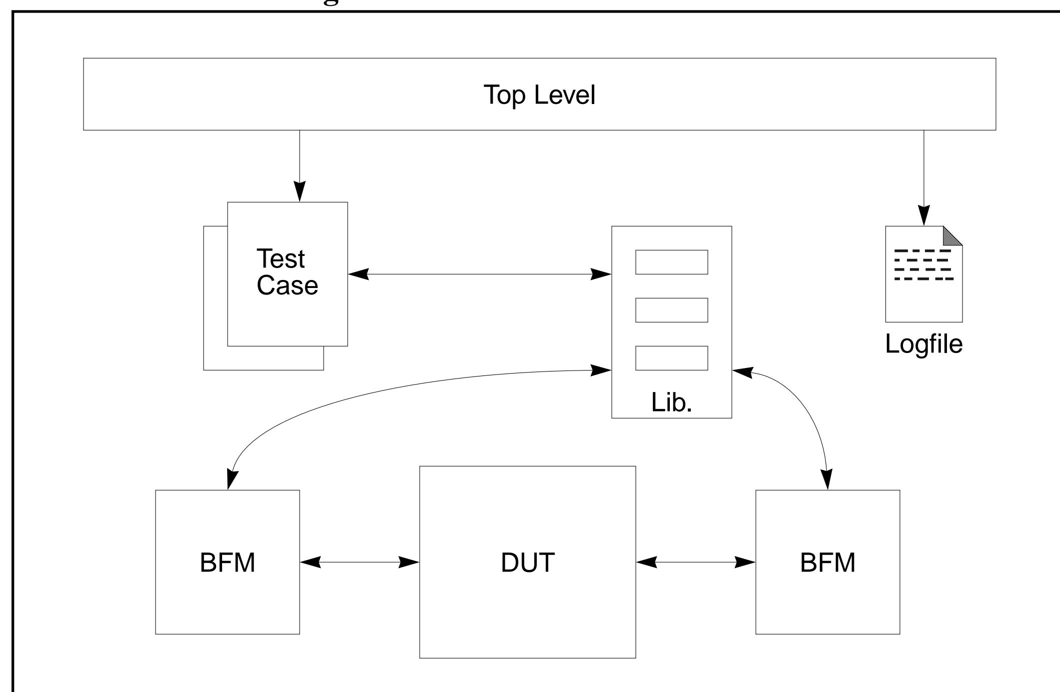
The general guidelines address the overall architecture of a test bench for individual building blocks (IP Cores) of a chip and an entire integrated circuit.

2.1. Reusability

In todays world of reusable IP Cores and constantly changing and improving technology, it is essential that an IP Cores test bench can be reused, as the specifications and technological features advance.

The test bench and all of its building blocks must be able to grow and allow the user to build on, new features and improvements as they become available. To achieve this goal, the verification engineer must very carefully plan the overall architecture of the verification environment. Below figure illustrates an example of such an environment.

Figure 1: Reusable Test Bench



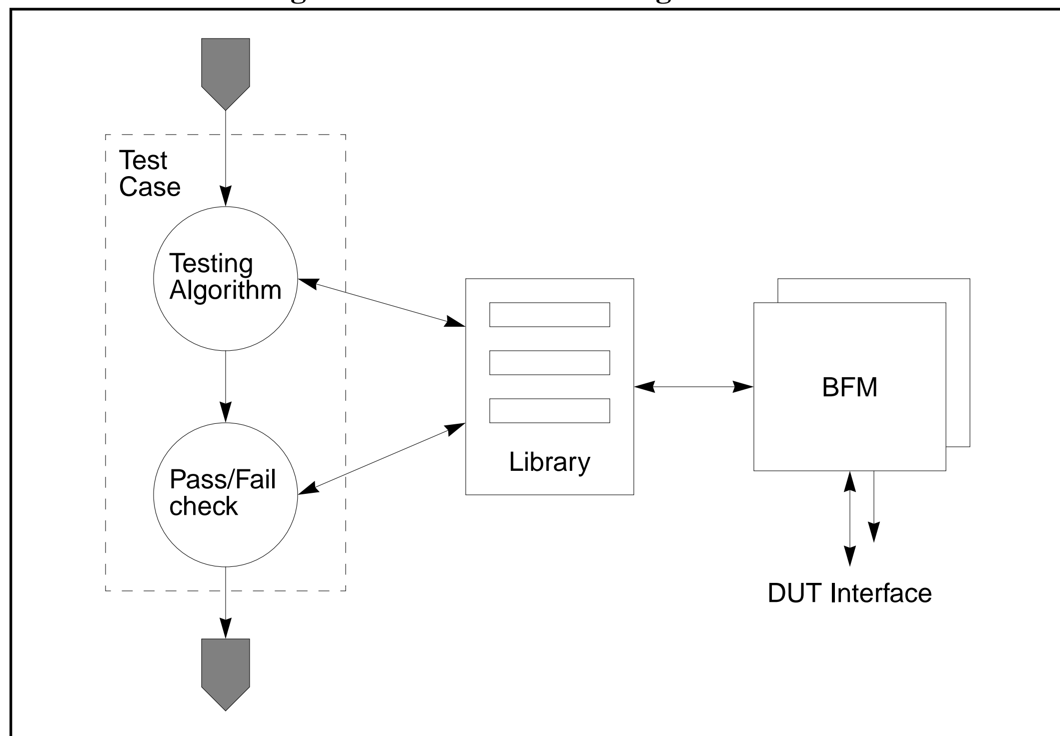
This sample verification environment consists of a Device Under Test (DUT), two Bus Functional Models (BFM), a library, several test cases and the all enclosing test bench.

2.2. Maximum Flexibility for Architectural Changes

To be able to add new feature and extend the building blocks of a test bench, one should define a clear interface between the various building blocks and information flow. As new features become available, and the DUT can perform additional functions, the previous test bench must be able to accommodate the new features and functions without being compleatly ewritten.

Lets take a look at a typical test case and it's flow.

Figure 2: Test Case Flow Diagram



Here we can clearly see, how the various blocks are interconnected.

2.2.1. Example

Lets analyze the above strategy with an example:

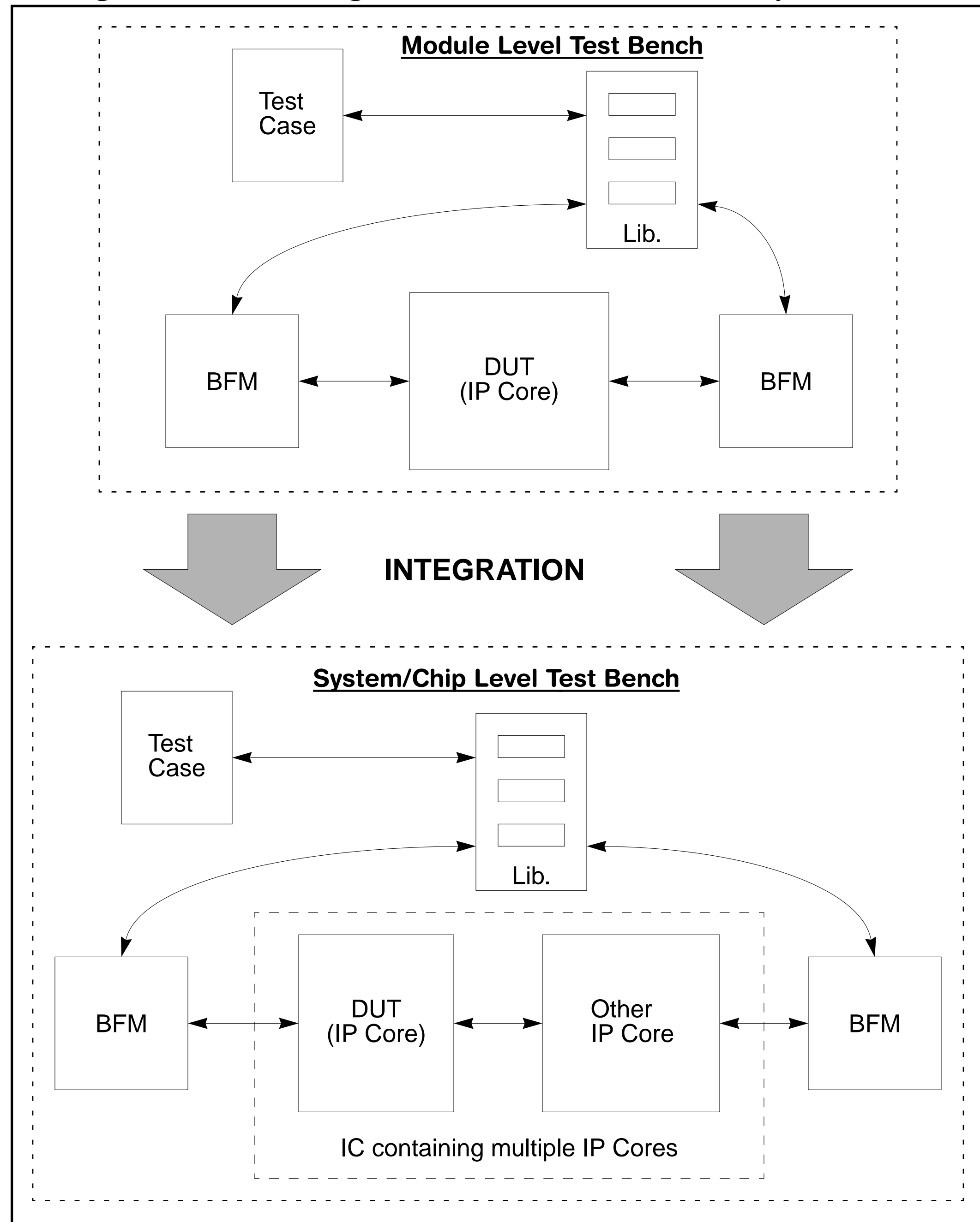
If our current device has a PCI 32 bit, 33Mhz interface, the BFM will emulate the PCI functions. The library will provide upper level functions. When the device is updated to PCI 64 bit, 66 Mhz, all that needs to be modified to get the existing test bench up and running, is the BFM. Additional task are then added to the library, and additional test cases can be written to test the new functionality. However, since PCI 64 bit, 66 Mhz is backwards compatible, we can use the existing test cases and library routines to test those functions. In other word only test cases

and library routines for new features need to be written, and the entire test bench can be reused.

2.3. Maximum Flexibility for IP Core Integration

The same strategy as in the previous section can also be applied for IP Cores that eventually will be integrated in to a larger chip.

Figure 3: IP Core Integration and Test Bench Reusability



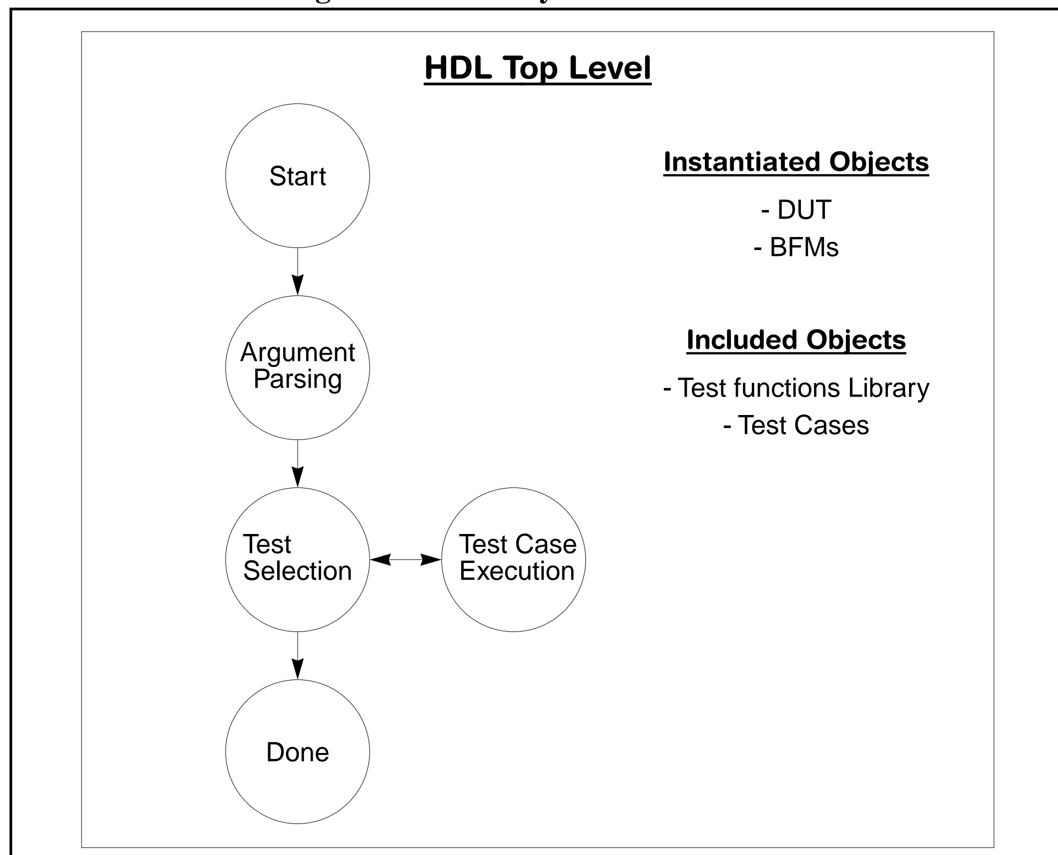
Here, one of the interfaces of the DUT, is now connected to another IP Core. Now most likely the BFM and associated library routines must be rewritten to provide the same functionality as in the module level test bench. After the BFM has been adjusted to the new interface, and the DUT instance has been updates, the entire test bench can be reused.

3

Test Bench

The test bench is the top level verification building block, that instantiates and includes all building blocks of the verification environment. It also should provide a way for the user to choose single test cases and provide a way to run all tests for regression testing. This is typically accomplished by including a few command line parsing statements that for example parse the plus arguments in Verilog-XL or compatible verilog simulator.

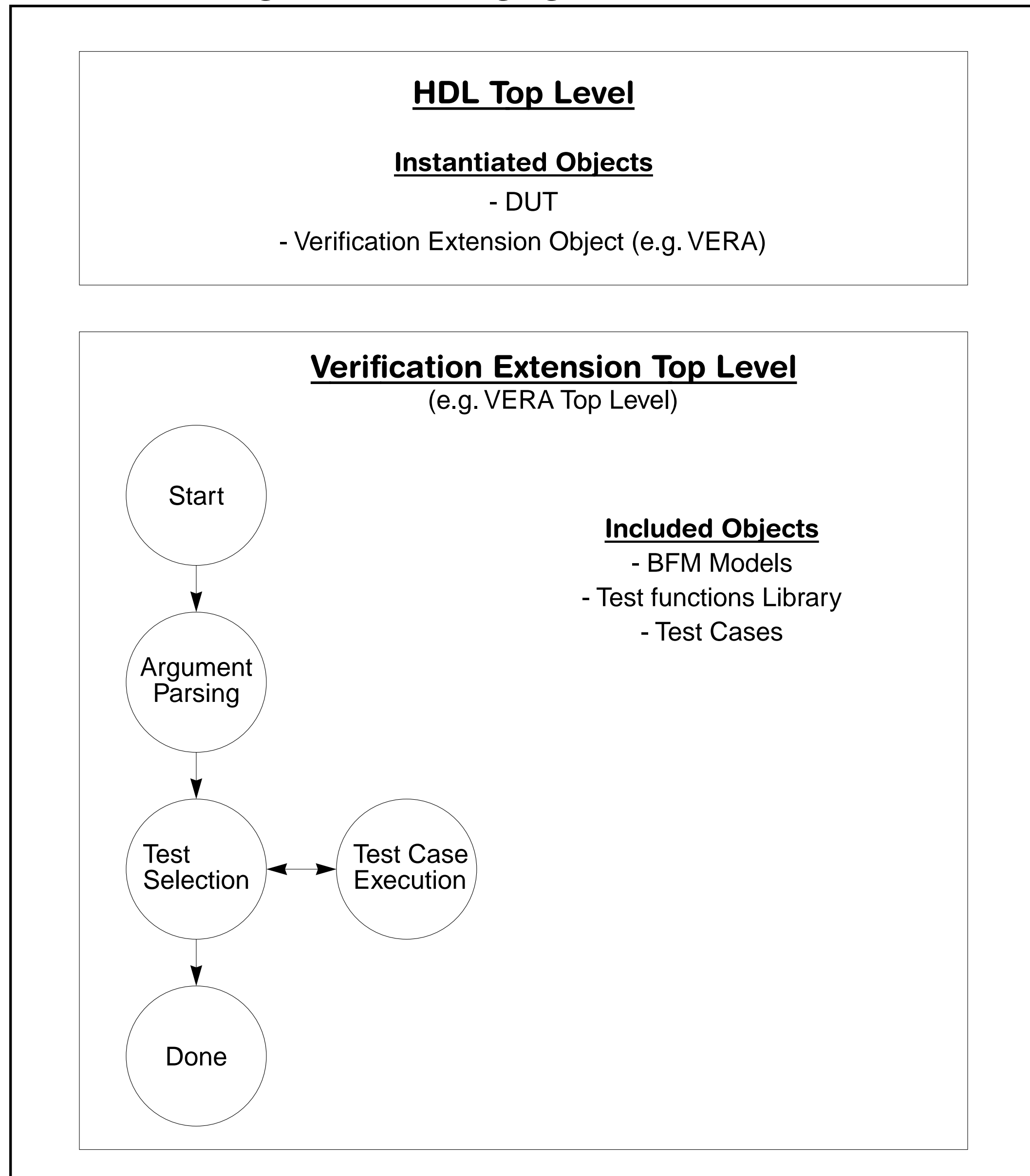
Figure 4: HDL only Test Bench



In some cases, there might be two top level test benches. Specifically this is the case when a separate verification language such as VERA from Synopsys is used.

In this case, the Verilog Top Level, instantiates the DUT and a VERA top level. The VERA top level instantiates the BFMs, library and test cases.

Figure 5: Multi Language Test Bench



3.1. Reusability Guidelines

Her are some pointers that should help in designing a test bench that can be easily reused.

- Partition the Test Bench
  Divide the test bench in to logically distinguished blocks:
  - Startup Section
    This is where the command line parsing and test case section is

    - POR & Clock section
      In this section all clocks and power on reset sequences are generated.
    - DUT section
      Here is where the DUT is instantiated
    - BFM section
      Here is where all BFMs go
- Create Modules for all functions
  For example, clock generation and POR should be contained in a module
- Create tasks/functions
  Avoid in-line code, move it instead to a task or function. For example command line argument parsing and test case section should in a task/function.
- Place all tasks/functions in to a test bench library.

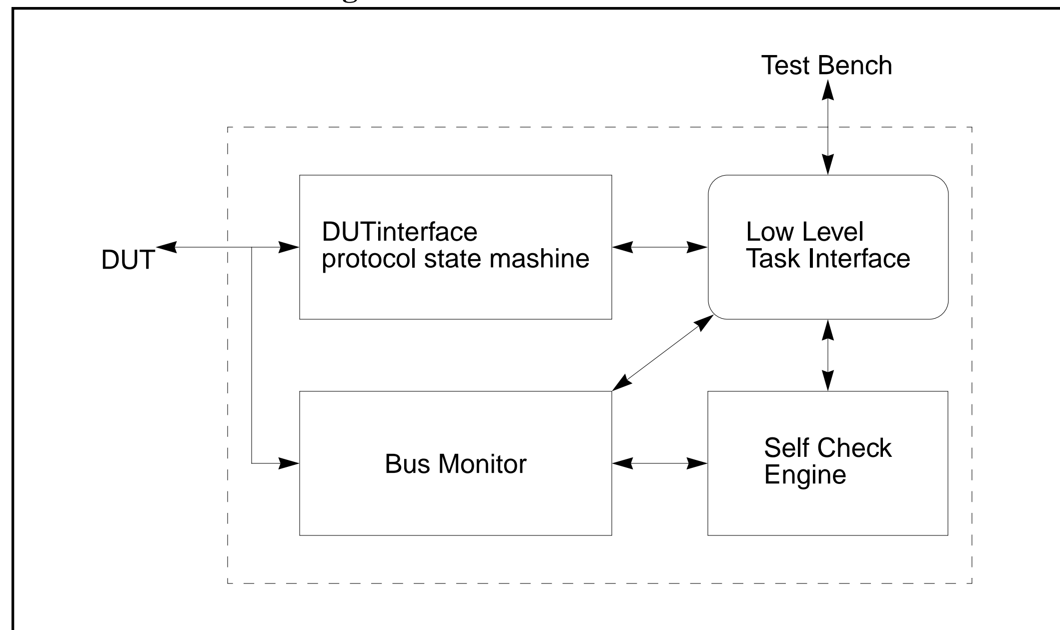
(This page intentionally left blank)

4

BFMs

Bus Functional Models (BFMs) create the iterface between a DUT and the test bench. The main prupose is to hide interface specific transaction and provide a low level generic task interface. This low level interface must be extreamly flexible and generic. It must allow to setup and perform illegal and out of bounds operation to check for proper erron handling of the DUT. Another very importend feature of BFM is to monitor proper interface operations from the DUT and provide some sort of self checking capabilities. The self checking capabilities become very importand when the BFM is the reciever of a transaction. It must provides a means to be informed what trabsaction to expect and another means to verify any associated data from the transaction.

Figure 6: BFM Architecture



4.1. Reusability Guidelines

Here are some pointers that should help in writing BFMs that can be reused.

- Partition the BFM in to logical sub modules
- Parameterize everything that might change in a feature release in the interface (e.g. bus width 32 -> 64 bits).
- Keep the test bench interface unrestricted, so that new parameters can be easily added.

5 Libraries

Libraries are an essential part of portable and reusable test benches. They contain higher level functions and operations that are required to perform specific tasks. Those operations can be as simple as the calculation of a CRC, and as complex as assembling IP packets or SONET frames.

It is essential, that all library functions are flexible and easy to use. The libraries should be divided in to separate files, for each building block of the test bench (e.g. each BFM should have it’s own library). Some common tasks and functions may be placed in to a shared library that is used across different building blocks and test benches.

(This page intentionally left blank)

6

Test Cases

Test Cases are the actual test programs that exercise a specific function of a DUT. They must be deterministic and provide a unambiguous, clear text status when they terminate. The preferred mechanism is to report “Test ABC PASSED” or “Test ABC FAILED in operation XYZ”.

In addition test cases must provide some sort of status while executing. There must be a way to determine if the simulation is hung or still running. This is not to imply that a test case should clobber the screen with lots of useless information, but that a test, that is known to run through, let’s say, 8 iteration that take 10 minutes to simulate each, will report which iteration is currently being simulated.

Each test case should provide a verbose execution mode, so in case of failure, it is easy to determine where the error occured.

When a failure is found, the test case can either recover from the error and continue to execute or abort the simulation. The actual action that is taken will depend on the test case. Test cases that can resume and continue, should allow for optional discontinuation of the current test case. In this case the test encvironment may advance to the next test case (if there any more available), or terminate the simulation run.

The test case writer, should try to keep the run time of test cases to some reasonable amount. Test cases that take days to complete are very difficult to debug, and unesessarily occupy CPU time when a test case is rerun because a bug that shows up after some 30 hours of simulation time has presumably been fixed. Many short test cases can be easely distributed accross sveral compute servers and will reduce the overall simulation time.

(This page intentionally left blank)

7

Miscellaneous

This section describes miscellaneous building blocks that can be used in a test bench. They provide additional functionality and might not be applicable in every case.

7.1. Simulation Reports and Log Files

It is strongly recommended that a test bench creates a log file, and logs progress to that file. Long simulation runs, consisting of many test cases, can be easily evaluated as to which test case has passed and which not. Sometimes screen buffers hold only a limited number of lines, and might loose important information. In addition, this will protect from complete reruns in case of power failures and computer crashes.

In addition, the test bench should provide a overall simulation report after all test cases have been executed. It is unreasonable to look though several megabytes of a text file, to try to determine if all test cases have passed.

7.2. Verbose and Quiet Simulations Modes

The test bench must provide a global flag that determines if a test bench runs in a quiet mode or in a verbose mode. Additional flags might be provided to determine the level of verbosity.

When initially debugging a DUT, it is important to have as much information as possible, about what is currently going on. After the initial debug phase, and as the DUT becomes more and more stable, the verbose level can be turned back. Finally as the DUT has been mostly debugged, and mostly regression testing is being performed, the quiet mode can be selected. This will not only limit the amount of information (some might consider it garbage) that is displayed on the screen, but will also increase the simulation time, as no (or very little) console IO is required.

7.3. Simulation Watchdog

Sometimes, it is essential to know when a simulation is hung in an endless loop, and it can't be easily determined if it is or not. In such cases it might be nec-

essary to build a simulation watchdog, which forces the simulation to stop, or provides some sort of notification to allow the operator to manually abort the simulation. The watchdog can be monitoring specific interfaces, that are know to have traffic or specific transaction. It is not recommended to reset the watchdog from test cases, as they sometimes can not determine when a DUT is hung.

7.4. Performance Monitors

In some cases, sustaining a certain performance level is part of verification. In others, it is just useful information that might help to improve the DUTs design. Performance monitors can be as simple a bus monitors that calculate the number of bytes per second, or as complex as IP packet monitors that provide an effective interface throughput.