Chapter 1
Verification Guidelines

Some believed we lacked the programming language to describe your perfect world…

(The Matrix, 1999)

Imagine that you are given the job of building a house for someone. Where should you begin? Do you start by choosing doors and windows, picking out paint and carpet colors, or selecting bathroom fixtures? Of course not! First you must consider how the owners will use the space, and their budget, so you can decide what type of house to build. Questions you should consider are: Do they enjoy cooking and want a high-end kitchen, or will they prefer watching movies in their home theater room and eating takeout pizza? Do they want a home office or an extra bedroom? Or does their budget limit them to a more modest house?

Before you start to learn details of the SystemVerilog language, you need to understand how you plan to verify your particular design and how this influences the testbench structure. Just as all houses have kitchens, bedrooms, and bathrooms, all testbenches share some common structure of stimulus generation and response checking. This chapter introduces a set of guidelines and coding styles for designing and constructing a testbench that meets your particular needs. These techniques use some of the same concepts that are shown in the *Verification Methodology Manual for SystemVerilog* (VMM), Bergeron et al. (2006), but without the base classes. Other methodologies such as UVM and OVM share the same concepts.

The most important principle you can learn as a verification engineer is: "Bugs are good." Don't shy away from finding the next bug, do not hesitate to ring a bell each time you uncover one, and furthermore, always keep track of the details of each bug found. The entire project team assumes there are bugs in the design, so each bug found before tape-out is one fewer that ends up in the customer's hands. At each stage in the design cycle such as specification, coding, synthesis, manufacturing, the cost of fixing a bug goes up by a factor of 10, so find those bugs early and often. You need to be as devious as possible, twisting and torturing the design to extract all possible bugs now, while they are still easy to fix. Don't let the designers steal all the glory — without your craft and cunning, the design might never work!

This book assumes you already know the Verilog language and want to learn the System Verilog Hardware Verification Language (HVL). Some of the typical features of an HVL that distinguish it from a Hardware Description Language such as Verilog or VHDL are:

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially Object-Oriented Programming, and transaction-level modeling
- Multi-threading and interprocess communication (IPC)
- Support for HDL types such as Verilog's 4-state values
- Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create testbenches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

1.1 The Verification Process

What is the goal of verification? If you answered, "Finding bugs," you are only partly correct. The goal of hardware design is to create a device that performs a particular task, such as a DVD player, network router, or radar signal processor, based on a design specification. Your purpose as a verification engineer is to make sure the device can accomplish that task successfully — that is, the design is an accurate representation of the specification. Bugs are what you get when there is a discrepancy. The behavior of the device when used outside of its original purpose is not your responsibility, although you want to know where those boundaries lie.

The process of verification parallels the design creation process. A designer reads the hardware specification for a block, interprets the human language description, and creates the corresponding logic in a machine-readable form, usually RTL code. To do this, he or she needs to understand the input format, the transformation function, and the format of the output. There is always ambiguity in this interpretation, perhaps because of ambiguities in the original document, missing details, or conflicting descriptions. As a verification engineer, you must also read the hardware specification, create the verification plan, and then follow it to build tests showing the RTL code correctly implements the features. Therefore, as a verification engineer, not only do you have to understand the design and its intent, but also, you have to consider all the corner test cases that the designer might not have thought about.

By having more than one person perform the same interpretation, you have added redundancy to the design process. As the verification engineer, your job is to read the same hardware specifications and make an independent assessment of what they mean. Your tests then exercise the RTL to show that it matches your interpretation.

1.1.1 Testing at Different Levels

What types of bugs are lurking in the design? The easiest ones to detect are at the block level, in modules created by a single person. Did the ALU correctly add two numbers? Did every bus transaction successfully complete? Did all the packets make it through a portion of a network switch? It is almost trivial to write directed tests to find these bugs, as they are contained entirely within one block of the design.

After the block level, the next place to look for discrepancies is at boundaries between blocks. This is known as the integration phase. Interesting problems arise when two or more designers read the same description yet have different interpretations. For a given protocol, what signals change and when? The first designer builds a bus driver with one view of the specification, while a second builds a receiver with a slightly different view. Your job is to find the disputed areas of logic and maybe even help reconcile these two different views.

To simulate a single design block, you need to create tests that generate stimuli from all the surrounding blocks — a difficult chore. The benefit is that these low-level simulations run very fast. However, you may find bugs in both the design and testbench, as the latter requires a great deal of code to provide stimuli from the missing blocks. As you start to integrate design blocks, they can stimulate each other, reducing your workload. These multiple block simulations may uncover more bugs, but they also run slower. Analyzing the behavior to determine the root cause of a bug is more time consuming at higher levels.

At the highest level of the Design Under Test (DUT), the entire system is tested, but the simulation performance is greatly reduced. Your tests should strive to have all blocks performing interesting activities concurrently. All I/O ports are active, processors are crunching data, and caches are being refilled. With all this action, data alignment and timing bugs are sure to occur.

At this level you are able to run sophisticated tests that have the DUT executing multiple operations concurrently so that as many blocks as possible are active. What happens if an MP3 player is playing music and the user tries to download new music from the host computer? Then, during the download, the user presses several of the buttons on the player? You know that when the real device is being used, someone is going to do all this, so why not try it out before it is built? This testing makes the difference between a product that is seen as easy to use and one that repeatedly locks up.

Once you have verified that the DUT performs its designated functions correctly, you need to see how it operates when there are errors. Can the design handle a partial transaction, or one with corrupted data or control fields? Just trying to enumerate all the possible problems is difficult, not to mention determining how the design should recover from them. Error injection and handling can be the most challenging part of verification.

As you move to system-level verification, the challenges also move to a higher level. At the block level, you can show that individual cells flow through the blocks of an ATM router correctly, but at the system level you might have to consider what happens if there are streams of different priority. Which cell should be chosen next is not always obvious at the highest level. You may have to analyze the statistics from thousands of cells to see if the aggregate behavior is correct.

One last point: you can never prove there are no bugs left, so you need to constantly come up with new verification tactics.

1.1.2 The Verification Plan

The verification plan is derived from the hardware specification and contains a description of what features need to be exercised and the techniques to be used. These steps may include directed or random testing, assertions, HW/SW co-verification, emulation, formal proofs, and use of verification IP. For a more complete discussion on verification see Bergeron (2006).

1.2 The Verification Methodology Manual

The book in your hands draws upon the VMM that has its roots in a methodology developed by Janick Bergeron and others at Qualis Design. They started with industry-standard practices and refined them based on their experience on many projects. VMM's techniques were originally developed for use with the OpenVera language and were extended in 2005 for SystemVerilog. VMM and its predecessor, the Reference Verification Methodology (RVM) for Vera, have been used successfully to verify a wide range of hardware designs, from networking devices to processors. Newer methodologies such as OVM and UVM use many similar ideas. This book is based on many of the same concepts as all these methodologies, though greatly simplified.

This book serves as a user guide for the SystemVerilog language. It describes many language constructs and provides guidelines for choosing the ones best suited to your needs. If you are new to verification, have little experience with Object-Oriented Programming (OOP), or are unfamiliar with constrained-random tests (CRT), this book can show you the right path to choose. Once you are familiar with them, you will find UVM and VMM to be an easy step up.

So why doesn't this book teach you UVM or VMM? Like any advanced tool, these methodologies were designed for use by an experienced user, and excel on difficult problems. Are you in charge of verifying a 100 million-gate design with many communication protocols, complex error handling, and a library of IP? If so, UVM or VMM are the right tools for the job. However, if you are working on smaller modules with a single protocol, you may not need such a robust methodology. Just remember that your block is part of a larger system; UVM- or VMM-compliant code is reusable both during a project and on later designs. The cost of verification goes beyond your immediate project.

The UVM and VMM have a set of base classes for data and environment, utilities for managing log files and interprocess communication, and much more. This book is an introduction to SystemVerilog and shows the techniques and tricks that go into these classes and utilities, giving you insight into their construction.

1.3 Basic Testbench Functionality

The purpose of a testbench is to determine the correctness of the DUT. This is accomplished by the following steps.

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness
- Measure progress against the overall verification goals

Some steps are accomplished automatically by the testbench, while others are manually determined by you. The methodology you choose determines how the preceding steps are carried out.

1.4 Directed Testing

Traditionally, when faced with the task of verifying the correctness of a design, you probably used directed tests. Using this approach, you look at the hardware specification and write a verification plan with a list of tests, each of which concentrated on a set of related features. Armed with this plan, you write stimulus vectors that exercise these features in the DUT. You then simulate the DUT with these vectors and manually review the resulting log files and waveforms to make sure the design does what you expect. Once the test works correctly, you check it off in the verification plan and move to the next one.

This incremental approach makes steady progress, which is always popular with managers who want to see a project making headway. It also produces almost immediate results, since little infrastructure is needed when you are guiding the creation of every stimulus vector. Given ample time and staffing, directed testing is sufficient to verify many designs.

Figure 1.1 shows how directed tests incrementally cover the features in the verification plan. Each test is targeted at a very specific set of design elements. If you had enough time, you could write all the tests needed for 100% coverage of the entire verification plan.

What if you do not have the necessary time or resources to carry out the directed testing approach? As you can see, while you may always be making forward progress, the slope remains the same. When the design complexity doubles, it takes twice as long to complete or requires twice as many people to implement it.

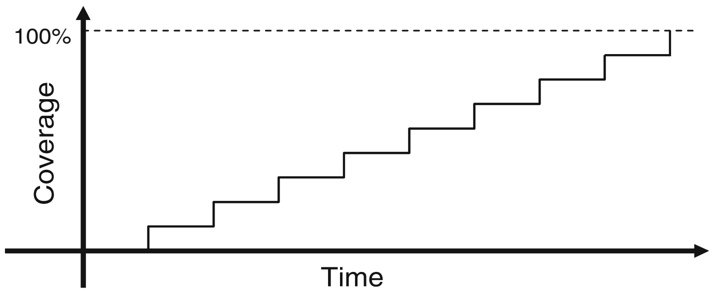

Fig. 1.1 Directed test progress over time

Neither of these situations is desirable. You need a methodology that finds bugs faster in order to reach the goal of 100% coverage. Brute force does not work; if you tried to verify every combination of inputs for a 32-bit adder, your simulations would still be running years after the project should have shipped.

Figure 1.2 shows the total design space and features that are covered by directed test cases. In this space there are many features, some of which have bugs. You need to write tests that cover all the features and find the bugs.

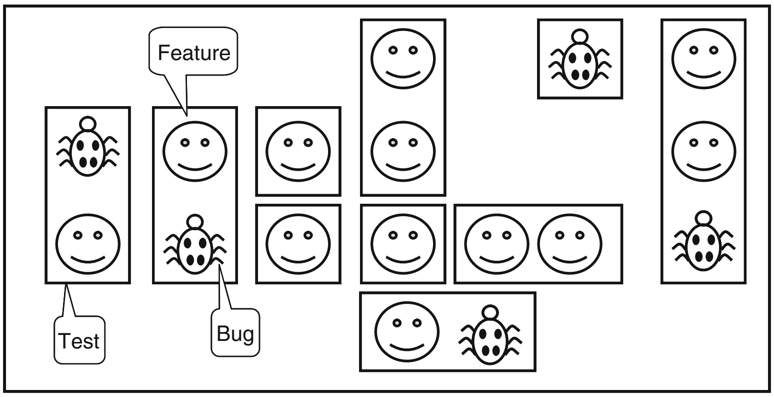

Fig. 1.2 Directed test coverage

1.5 Methodology Basics

This book uses the following principles.

- Constrained-random stimulus
- Functional coverage
- Layered testbench using transactors
- Common testbench for all tests
- Test case-specific code kept separate from testbench

All these principles are related. Random stimulus is crucial for exercising complex designs. A directed test finds the bugs you expect to be in the design, whereas a random test can find bugs you never anticipated. When using random stimuli, you need functional coverage to measure verification progress. Furthermore, once you start using automatically generated stimuli, you need an automated way to predict the results — generally a scoreboard or reference model. Building the testbench infrastructure, including self-prediction, takes a significant amount of work. A layered testbench helps you control the complexity by breaking the problem into manageable pieces. Transactors provide a useful pattern for building these pieces. With appropriate planning, you can build a testbench infrastructure that can be shared by all tests and does not have to be continually modified. You just need to leave "hooks" where the tests can perform certain actions such as shaping the stimulus and injecting disturbances. Conversely, code specific to a single test must be kept separate from the testbench to prevent it from complicating the infrastructure.

Building this style of testbench takes longer than a traditional directed testbench — especially the self-checking portions. As a result, there may be a significant delay before the first test can be run. This gap can cause a manager to panic, so make this effort part of your schedule. In Fig. 1.3, you can see the initial delay before the first random test runs.

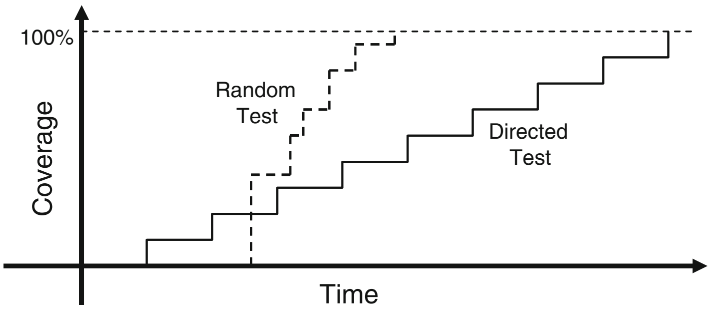

Fig. 1.3 Constrained-random test progress over time vs. directed testing

While this up-front work may seem daunting, the payback is high. Every random test you create shares this common testbench, as opposed to directed tests where each is written from scratch. Each random test contains a few dozen lines of code to constrain the stimulus in a certain direction and cause any desired exceptions, such as creating a protocol violation. The result is that your single constrained-random testbench is now finding bugs faster than the many directed ones.

As the rate of discovery begins to drop off, you can create new random constraints to explore new areas. The last few bugs may only be found with directed tests, but the vast majority of bugs will be found with random tests. If you create a random testbench, you can always constrain it to created directed tests, but a directed testbench can never be turned into a true random testbench.

1.6 Constrained-Random Stimulus

Although you want the simulator to generate the stimulus, you don't want totally random values. You use the SystemVerilog language to describe the format of the stimulus ("address is 32-bits; opcode is ADD, SUB or STORE; length < 32 bytes"), and the simulator picks values that meet the constraints. Constraining the random values to become relevant stimuli is covered in Chapter 6. These values are sent into the design, and are also sent into a high-level model that predicts what the result should be. The design's actual output is compared with the predicted output.

Figure 1.4 shows the coverage for constrained-random tests over the total design space. First, notice that a random test often covers a wider space than a directed one. This extra coverage may overlap other tests, or may explore new areas that you did not anticipate. If these new areas find a bug, you are in luck! If the new area is not legal, you need to write more constraints to keep random generation from creating illegal design functionality. Lastly, you may still have to write a few directed tests to find cases not covered by any other constrained-random tests.

Figure 1.5 shows the paths to achieve complete coverage. Start at the upper left with basic constrained-random tests. Run them with many different seeds.

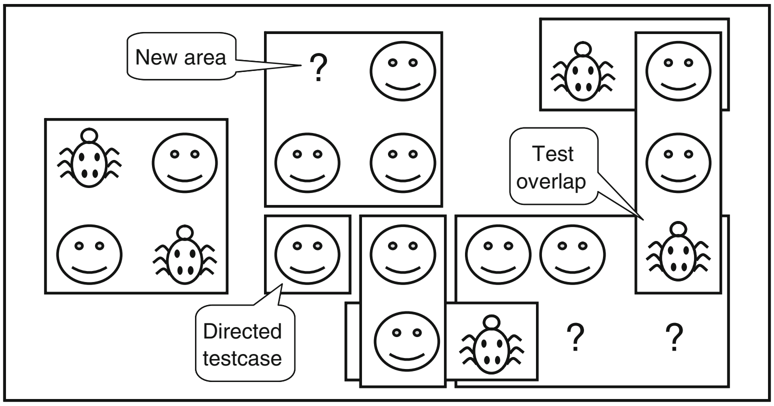

Fig. 1.4 Constrained-random test coverage

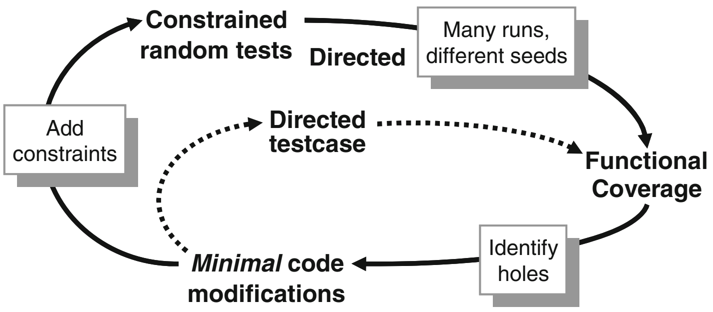

Fig. 1.5 Coverage convergence

When you look at the functional coverage reports, find the holes where there are gaps in the coverage. Then you make minimal code changes, perhaps by using new constraints, or by injecting errors or delays into the DUT. Spend most of your time in this outer loop, writing directed tests for only the few features that are very unlikely to be reached by random tests.

1.7 What Should You Randomize?

When you think of randomizing the stimulus to a design, you might first pick the data fields. These values are is the easiest to create — just call `$random ()`. The problem is that this choice gives a very low payback in terms of bugs found. The primary types of bugs found with random data are data path errors, perhaps with bit-level mistakes. You need to find bugs in the control logic, source of the most devious problems.

Think broadly about all design inputs, such as the following.

- Device configuration
- Environment configuration
- Input data
- Protocol exceptions
- Errors and violations
- Delays

These are discussed in sections 1.7.1 through 1.7.4.

1.7.1 Device and Environment Configuration

What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations are tried. Most tests just use the design as it comes out of reset, or apply a fixed set of initialization vectors to put it into a known state. This is like testing a PC's operating system right after it has been installed, but without any of the applications installed. Of course the performance is fine and there aren't any crashes.

In a real world environment, the DUT's configuration becomes more random the longer it is in use. For example, I helped a company verify a time-division multiplexor switch that had 2000 input channels and 12 output channels. The verification engineer said, "These channels could be mapped to various configurations on the other side. Each input could be used as a single channel, or further divided into multiple channels. The tricky part is that although a few standard ways of breaking it down are used most of the time, any combination of breakdowns is legal, leaving a huge set of possible customer configurations."

To test this device, the engineer had to write several dozen lines of directed testbench code to configure each channel. As a result, she was never able to try configurations with more than a handful of channels. Together, we wrote a testbench that randomized the parameters for a single channel and then put this in a loop to configure all the switch's channels. Now she had confidence that her tests would uncover configuration-related bugs that would have been missed before.

In the real world, your device operates in an environment containing other components. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment configuration, including the length of the simulation, number of devices, and how they are configured. Of course you need to create constraints to make sure the configuration is legal.

In another Synopsys customer example, a company created an I/O switch chip that connected multiple PCI buses to an internal memory bus. At the start of simulation they randomly chose the number of PCI buses (1–4), the number of devices on each bus (1–8), and the parameters for each device (master or slave, CSR addresses, etc.). They kept track of the tested combinations using functional coverage so that they could be sure that they had covered almost every possible one.

Other environment parameters include test length, error injection rates, and delay modes. See Bergeron (2006) for more examples.

1.7.2 Input Data

When you read about random stimulus, you probably thought of taking a transaction such as a bus write or ATM cell and filling the data fields with random values. Actually, this approach is fairly straightforward as long as you carefully prepare your transaction classes as shown in Chapters 5 and 8. You need to anticipate any layered protocols and error injection, plus scoreboarding and functional coverage.

1.7.3 Protocol Exceptions, Errors, and Violations

There are few things more frustrating than when a device such as a PC or cell phone locks up. Many times, the only cure is to shut it down and restart. Chances are that deep inside the product there is a piece of logic that experienced some sort of error condition from which it could not recover and thus prevented the device from working correctly.

How can you prevent this from happening to the hardware you are building? If something can go wrong in the real hardware, you should try to simulate it. Look at all the errors that can occur. What happens if a bus transaction does not complete? If an invalid operation is encountered? Does the design specification state that two signals are mutually exclusive? Drive them both and make sure the device continues to operate properly.

Just as you are trying to provoke the hardware with ill-formed commands, you should also try to catch these occurrences. For example, recall those mutually exclusive signals. You should add checker code to look for these violations. Your code should at least print a warning message when this occurs, and preferably generate an error and wind down the test. It is frustrating to spend hours tracking back through code trying to find the root of a malfunction, especially when you could have caught it close to the source with a simple assertion. See Vijayaraghavan [2005] for more guidelines on writing assertions in your testbench and design code. Just make sure that you can disable the code that stops simulation on error so that you can easily test error handling.

1.7.4 Delays and Synchronization

How fast should your testbench send in stimulus? You should pick random delays to help catch protocol bugs. A test with the shortest delays is easy to write, but won't create all possible stimulus combinations. Subtle bugs around boundary conditions are often revealed when realistic delays are chosen.

A block may function correctly for all possible permutations of stimulus from a single interface, but subtle errors may occur when transactions are flowing into multiple inputs. Try to coordinate the various drivers so they can communicate at different timing rates. What if the inputs arrive at the fastest possible rate, but the output is being throttled back to a slower rate? What if stimulus arrives at multiple inputs concurrently? What if it is staggered with different delays? Use functional coverage, which will be discussed in Chapter 9, to measure what combinations have been randomly generated.

1.7.5 Parallel Random Testing

How should you run the tests? A directed test has a testbench that produces a unique set of stimulus and response vectors. To change the stimulus, you need to change the test. A random test consists of the testbench code plus a random seed. If you run the same test 50 times, each time with a unique seed, you will get 50 different sets of stimuli. Running with multiple seeds broadens the coverage of your test and leverages your work.

You need to choose a unique seed for each simulation. Some people use the time of day, but that can still cause duplicates. What if you are using a batch queuing system across a CPU farm and tell it to start 10 jobs at midnight? Multiple jobs could start at the same time but on different computers, and will thus get the same random seed and run the same stimulus. You should blend in the processor name to the seed. If your CPU farm includes multiprocessor machines, you could have two jobs start running at midnight with the same seed, so you should also throw in the process ID. Now all jobs get unique seeds.

You need to plan how to organize your files to handle multiple simulations. Each job creates a set of output files, such as log files and functional coverage data. You can run each job in a different directory, or you can try to give a unique name to each file. The easiest approach is to append the random seed value to the directory name.

1.8 Functional Coverage

Sections 1.6 and 1.7 showed how to create stimuli that can randomly walk through the entire space of possible inputs. With this approach, your testbench visits some areas often, but takes too long to reach all possible states. Unreachable states will never be visited, even given unlimited simulation time. You need to measure what has been verified in order to check off items in your verification plan.

The process of measuring and using functional coverage consists of several steps. First, you add code to the testbench to monitor the stimulus going into the device, and its reaction and response, to determine what functionality has been exercised. Run several simulations, each with a different seed. Next, merge the results from these simulations into a report. Lastly, you need to analyze the results and determine how to create new stimulus to reach untested conditions and logic. Chapter 9 describes functional coverage in SystemVerilog.

1.8.1 Feedback from Functional Coverage to Stimulus

A random test evolves using feedback. The initial test can be run with many different seeds, thus creating many unique input sequences. Eventually the test, even with a new seed, is less likely to generate stimulus that reaches areas of the design space. As the functional coverage asymptotically approaches its limit, you need to change the test to find new approaches to reach uncovered areas of the design. This is known as "coverage-driven verification" and is shown in Fig. 1.6.

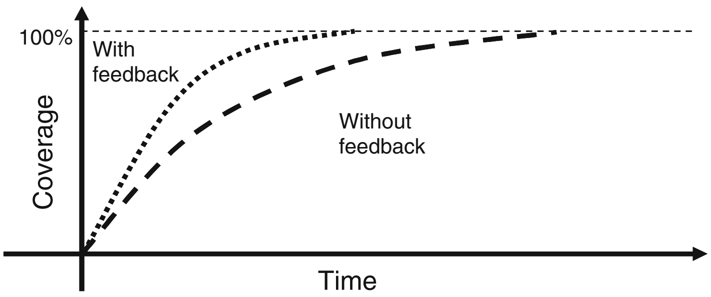

Fig. 1.6 Test progress with and without feedback

What if your testbench were smart enough to do this for you? In a previous job, I wrote a test that generated every bus transaction for a processor and additionally fired every bus terminator (Success, Parity Error, Retry) in every cycle. This was before HVLs, so I wrote a long set of directed tests and spent days lining up the terminator code to fire at just the right cycles. After much hand analysis I declared success — 100% coverage. Then the processor's timing changed slightly! Now I had to reanalyze the test and change the stimuli.

A more productive testing strategy uses random transactions and terminators. The longer you run it, the higher the coverage. As a bonus, the test can be made flexible enough to create valid stimuli even if the design's timing changed. You can accomplish this by adding a feedback loop that looks at the stimulus created so far (generated all write cycles yet?) and then change the constraint weights (drop write weight to zero). This improvement would greatly reduce the time needed to get to full coverage, with little manual intervention.

This is not a typical situation however, because of the trivial feedback from functional coverage to the stimulus. In a real design, how should you change the stimulus to reach a desired design state? This requires deep knowledge of the design and powerful formal techniques. There are no easy answers, so dynamic feedback is rarely used for constrained-random stimulus. Instead, you need to manually analyze the functional coverage reports and alter your random constraints.

Feedback is used in formal analysis tools such as Magellan (Synopsys 2003). It analyzes a design to find all the unique, reachable states. It then runs a short simulation to see how many states were visited. Lastly, it searches from the state machine to the design inputs to calculate the stimulus needed to reach any remaining states and then Magellan applies this to the DUT.

1.9 Testbench Components

In simulation, the testbench wraps around the DUT, just as a hardware tester connects to a physical chip, as shown in Fig. 1.7. Both the testbench and tester provide stimulus and capture responses. The difference between them is that your testbench needs to work over a wide range of levels of abstraction, creating transactions and sequences, which are eventually transformed into bit vectors. A tester just works at the bit level.

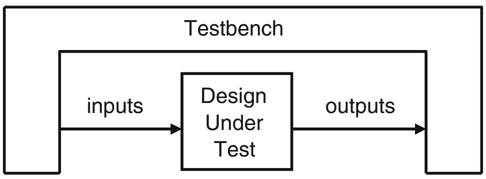

Fig. 1.7 The testbench — design environment

What goes into that testbench block? It is comprised of many Bus Functional Models (BFM), which you can think of as testbench components — to the DUT they look like real components, but they are part of the testbench, not the RTL design. If the real device connects to AMBA, USB, PCI, and SPI buses, you have to build equivalent components in your testbench that can generate stimulus and check the response, as shown in Fig. 1.8. These are not detailed, synthesizable models, but instead highlevel transactors that obey the protocol, and execute more quickly. On the other hand, if you are prototyping using FPGAs or emulation, the BFMs do need to be synthesizable.

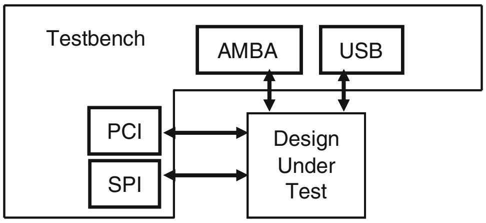

Fig. 1.8 Testbench components

1.10 Layered Testbench

A key concept for any modern verification methodology is the layered testbench. Although this process may seem to make the testbench more complex, it actually helps to make your task easier by dividing the code into smaller pieces that can be developed separately. Don't try to write a single routine that can randomly generate all types of stimuli, both legal and illegal, plus inject errors with a multi-layer protocol. The routine quickly becomes complex and unmaintainable. In addition, a layered approach allows reuse and encapsulation of Verification IP (VIP) which are OOP concepts.

1.10.1 A Flat Testbench

When you first learned Verilog and started writing tests, they probably looked like the low-level code in Sample 1.1, which does a simplified APB (AMBA Peripheral Bus) Write. (VHDL users may have written similar code).

Sample 1.1 Driving the APB pins

```
module test(PAddr, PWrite, PSel, PWData, PEnable, Rst, clk);
// Port declarations omitted...

  initial begin
    // Drive reset
    Rst <= 0;
    #100 Rst <= 1'b1;

    // Drive the control bus
    @(posedge clk)
    PAddr  <= 16'h50;
    PWData <= 32'h50;
    PWrite <= 1'b1;
    PSel   <= 1'b1;

    // Toggle PEnable
    @(posedge clk)
      PEnable <= 1'b1;
    @(posedge clk)
      PEnable <= 1'b0;

    // Check the result
    if (top.mem.memory[16'h50] == 32'h50)
      $display("Success");
    else
      $display("Error, wrong value in memory");
    $finish;
  end
endmodule
```

After a few days of writing code like this, you probably realized that it is very repetitive, so you created tasks for common operations such as a bus write, as shown in Sample 1.2.

Sample 1.2 A task to drive the APB pins

```
task write(reg [15:0] addr, reg [31:0] data);
  // Drive Control bus
  @(posedge clk)
  PAddr  <= addr;
  PWData <= data;
  PWrite <= 1'b1;
  PSel   <= 1'b1;

  // Toggle Penable
  @(posedge clk)
    PEnable <= 1'b1;
  @(posedge clk)
    PEnable <= 1'b0;
endtask
```

Now your testbench became simpler, as shown in Sample 1.3

Sample 1.3 Low-level Verilog test

```
module test(PAddr,PWrite,PSel,PWData,PEnable,Rst,clk);
  // Port declarations omitted...

  // Tasks as shown in Sample 1-2

  initial begin
    reset();                 // Reset the device
    write(16'h50, 32'h50);   // Write data into memory

    // Check the result
    if (top.mem.memory[16'h50] == 32'h50)
      $display("Success");
    else
      $display("Error, wrong value in memory");
    $finish;
  end
endmodule
```

By taking the common actions (such as reset, bus reads and writes) and putting them in a routine, you became more efficient and made fewer mistakes. This creation of the physical and command layers is the first step to a layered testbench.

1.10.2 The Signal and Command Layers

Figure 1.9 shows the lower layers of a testbench.

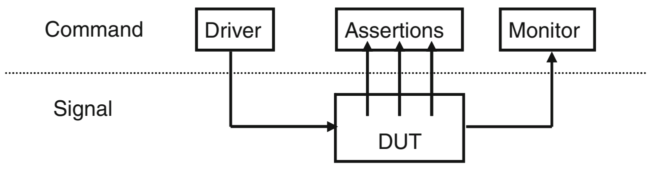

Fig. 1.9 Signal and command layers

At the bottom is the signal layer that contains the design under test and the signals that connect it to the testbench.

The next higher level is the command layer. The DUT's inputs are driven by the driver that runs single commands, such as bus read or write. The DUT's output drives the monitor that takes signal transitions and groups them together into commands. Assertions also cross the command/signal layer, as they look at individual signals and also changes across an entire command.

1.10.3 The Functional Layer

Figure 1.10 shows the testbench with the functional layer added, which feeds down into the command layer. The agent block (called the transactor in the VMM) receives higher-level transactions such as DMA read or write and breaks them into individual commands or transactions. These commands are also sent to the scoreboard that predicts the results of the transaction. The checker compares the commands from the monitor with those in the scoreboard.

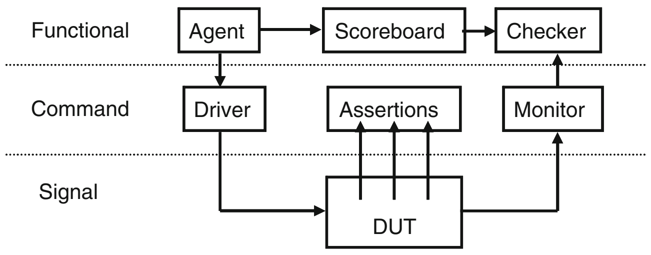

Fig. 1.10 Testbench with functional layer added

1.10.4 The Scenario Layer

The functional layer is driven by the generator in the scenario layer, as shown in Fig. 1.11. What is a scenario? Remember that your job as a verification engineer is to make sure that this device accomplishes its intended task. An example device is an MP3 player that can concurrently play music from its storage, download new music from a host, and respond to input from the user, such as adjusting the volume and track controls. Each of these operations is a scenario. Downloading a music file takes several steps, such as control register reads and writes to set up the operation, multiple DMA writes to transfer the song, and then another group of reads and writes. The scenario layer of your testbench orchestrates all these steps with constrained-random values for parameters such as track size and memory location.

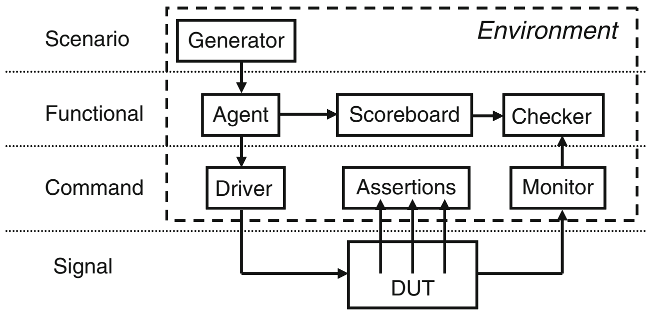

Fig. 1.11 Testbench with scenario layer added

The blocks in the testbench environment (inside the dashed line of Fig. 1.11) are written at the beginning of development. During the project they may evolve and you may add functionality, but these blocks should not change for individual tests. This is done by leaving “hooks” in the code so that a test can change the behavior of these blocks without having to rewrite them. You create these hooks with factory patterns (Section 8.2) and callbacks (Section 8.7).

1.10.5 The Test Layer and Functional Coverage

You are now at the top of the testbench, in the test layer, as shown in Fig. 1.12. Design bugs that occur between DUT blocks are harder to find as they involve multiple people reading and interpreting multiple specifications.

This top-level test is the conductor: he does not play any musical instrument, but instead guides the efforts of others. The test contains the constraints to create the stimulus.

Functional coverage measures the progress of all tests in fulfilling the verification plan requirements. The functional coverage code changes through the project as the various criteria complete. This code is constantly being modified and thus it is not part of the environment.

You can create a directed test in a constrained-random environment. Simply insert a section of directed test code into the middle of a random sequence, or put the two pieces of code in parallel. The directed code performs the work you want, but the random "background noise" may cause a bug to become visible, perhaps in a block that you never considered.

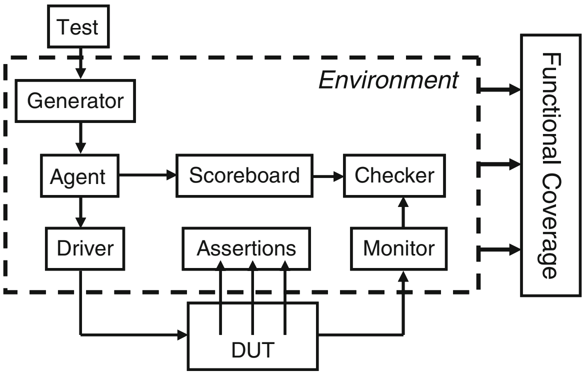

Fig. 1.12 Full testbench with all layers

Do you need all these layers in your testbench? The answer depends on what your DUT looks like. A complicated design requires a sophisticated testbench. You always need the test layer. For a simple design, the scenario layer may be so simple that you can merge it with the agent. When estimating the effort to test a design, don't count the number of gates; count the number of designers. Every time you add another person to the team, you increase the chance of different interpretations of the specifications. Typical hardware teams need more than two verification engineers for every designer.

You may need more layers. If your DUT has several protocol layers, each should get its own layer in the testbench environment. For example, if you have TCP traffic that is wrapped in IP and sent in Ethernet packets, consider using three separate layers for generation and checking. Better yet, use existing verification components.

One last note about Fig. 1.12. It shows some of the possible connections between blocks, but your testbench may have a different set. The test may need to reach down to the driver layer to force physical errors. What has been described here is just guidelines — let your needs guide what you create.

1.11 Building a Layered Testbench

Now it is time to take the preceding figures and learn how to map the components into SystemVerilog constructs.

1.11.1 Creating a Simple Driver

First, take a closer look at one of the blocks, the driver.

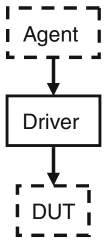

Fig. 1.13 Connections for the driver

The driver shown in Fig. 1.13 receives commands from the agent. The driver may inject errors or add delays. It then breaks the command down into individual signal changes such as bus requests and handshakes. The general term for such a testbench block is a "transactor," which, at its core, is a loop. Sample code for a transactor is shown in Sample 1.4.

Sample 1.4 Basic transactor code

```
task run();
  done = 0;
  while (!done) begin
    // Get the next transaction
    // Make transformations
    // Send out transactions
  end
endtask
```

Chapter 5 presents basic OOP and how to create an object that includes the routines and data for a transactor. Another example of a transactor is the agent. It might break apart a complex transaction such as a DMA read into multiple bus commands. Also in Chapter 5, you will see how to build an object that contains the data and routines that make up a command. These objects are sent between transactors using SystemVerilog mailboxes. In Chapter 7, you will learn about many ways to exchange data between the different layers and to synchronize the transactors.

1.12 Simulation Environment Phases

Up until now you have been learning what parts make up the environment. When do these parts execute? You want to clearly define the phases to coordinate the testbench so that all the code for a project works together. The three primary phases are Build, Run, and Wrap-up. Each is divided into smaller steps. These three are a subset of the many phases of the UVM and VMM.

The Build phase is divided into the following steps:

- *Generate configuration*: Randomize the configuration of the DUT and the surrounding environment.
- *Build environment*: Allocate and connect the testbench components based on the configuration. A testbench component is one that only exists in the testbench, as opposed to physical components in the design that are built with RTL code. For example, if the configuration chose three bus drivers, the testbench would allocate and initialize them in this step.
- *Reset the DUT*.
- *Configure the DUT*: Based on generated configuration from the first step, load the DUT command registers.

The Run phase is where the test actually runs. It has the following steps:

- *Start environment*: Run the testbench components such as BFMs and stimulus generators.
- *Run the test*: Start the test and then wait for it to complete. It is easy to tell when a directed test has completed, but doing so can be complex for a random test. You can use the testbench layers as a guide. Starting from the top, wait for a layer to drain all the inputs from the previous layer (if any), wait for the current layer to become idle, and then wait for the next lower layer. You should also use time-out checkers to ensure that the DUT or testbench does not lock up.

The Wrap-up phase has two steps:

- *Sweep*: After the lowest layer completes, you need to wait for the final transactions to drain out of the DUT.
- *Report*: Once the DUT is idle, sweep the testbench for lost data. Sometimes the scoreboard holds transactions that never came out, perhaps because they were dropped by the DUT. Armed with this information, you can create the final report on whether the test passed or failed. If it failed, be sure to delete any functional coverage results, as they may not be correct.

As shown in Fig. 1.12, the test starts the environment, which, in turn, runs each of the steps. More details can be found in Chapter 8.

1.13 Maximum Code Reuse

To verify a complex device with hundreds of features, you have to write hundreds of directed tests. If you use constrained-random stimulus, you would write fewer tests. Instead, the real work is put into constructing the testbench, which contains all the lower testbench layers: scenario, functional, command, and signal. This testbench code is used by all the tests, so it remains generic.

These guidelines appear to recommend an overly complicated testbench, but remember that every line that you put into a testbench can eliminate a line in every single test. If you know you will be creating a few dozen tests, there is a high payback in making a more sophisticated testbench. Keep this in mind when you read Chapter 8.

1.14 Testbench Performance

If this is the first time you have seen this methodology, you probably have some qualms about how it works compared to directed testing. A common objection is testbench performance. A directed test often simulates in a few seconds, whereas constrained-random tests will wander around through the state space for minutes or even hours. The problem with this argument is that it ignores a real verification bottleneck: the time required by you to create a test. You may be able to hand-craft a directed test in a day and debug it and manually verify the results by hand in another day or two. The actual simulation run time is dwarfed by the amount of time that you personally invested.

There are several steps to creating a constrained-random test. The first and most significant step is building the layered testbench, including the self-checking portion. The benefit of this work is shared by all tests, so it is well worth the effort. The second step is creating the stimulus specific to a goal in the verification plan. You may be crafting random constraints, or devious ways of injecting errors or protocol violations. Building one of these may take more time than making several directed tests, but the payoff will be much higher. A constrained-random test that tries thousands of different protocol variations is worth more than the handful of directed tests that could have been created in the same amount of time.

The third step in constrained-random testing is functional coverage. This task starts with the creation of a strong verification plan with clear goals that can be easily measured. Next you need to create the SystemVerilog code that adds instrumentation to the environment and gathers the data. Finally, it is essential that you analyze the results to determine if you have met the goals, and, if not, how you should modify the tests.

1.15 Conclusion

The continuous growth in complexity of electronic designs requires a modern, systematic, and automated approach to creating testbenches. The cost of fixing a bug grows by tenfold as a project moves from each step of specification to RTL coding, gate synthesis, fabrication, and finally into the user's hands. Directed tests only test one feature at a time and cannot create the complex stimulus and configurations that the device would be subjected to in the real world. To produce robust designs, you must use constrained-random stimulus combined with functional coverage to create the widest possible range of stimuli.

1.16 Exercises

1. Write a verification plan for an Arithmetic Logic Unit (ALU) with:
    - Asynchronous active high input reset
    - Input clock
    - 4-bit signed inputs, A and B
    - 5-bit signed output C that is registered on the positive edge of input clock.
    - 4 opcodes
        - Add: A + B
        - Sub: A – B
        - Bit-wise invert: A
        - Reduction Or: B
2. What are the advantages and disadvantages to testing at the block level? Why?
3. What are the advantages and disadvantages to testing at the system level? Why?
4. What are the advantages and disadvantages to directed testing? Why?
5. What are the advantages and disadvantages to constrained random testing? Why?