CHAPTER 5 STIMULUS AND RESPONSE

The purpose of writing testbenches is to apply stimulus signals to a design and observe the response. That response must then be compared against the expected behavior.

This chapter shows how to apply stimulus and observe response.

In this chapter, I show how to generate the stimulus signals. The greatest challenge with stimulus is making sure they are an accurate representation of the environment, not just a simple case. In this chapter I also show how to observe response, and more importantly, how to compare it against expected values. The final part of this chapter covers techniques for communicating the predicted the output to the monitors.

The next chapter shows how to structure a testbench.

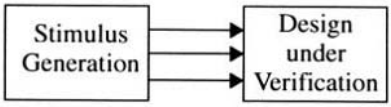
In the next chapter, I show how to best structure the stimulus generators and response monitors and the testcases that use them to minimize maintenance, and increase reusability across testbenches. If you prefer a top-down perspective, I recommend you start with the next chapter then come back to this one.

SIMPLE STIMULUS

In this section, I explain how to generate deterministic waveforms. Various techniques are developed to best generate stimulus signals. I show how synchronized waveforms can be properly generated and how to avoid underconstraining stimulus. I also demonstrate how to encapsulate and package signal generation operations using bus-functional models.

Generating stimulus is the process of providing input signals to the design under verification as shown in Figure 5-1. From the perspective of the stimulus generator, every input of the design is an output of the generator.

Figure 5-1. Stimulus generation



Generating a Simple Waveform

Clock signals are simple signals, but must be generated with care.

Because a clock signal has a very simple repetitive pattern, it is one of the first and most fundamental signals to generate. It is also the most critical signal to generate accurately. Many other signals use the clock signal to synchronize themselves.

The behavioral code to generate a 50 percent duty-cycle 100MHz clock signal is shown in Sample 5-1. To produce a more robust clock generator, use explicit assignments of values '0' and '1'. Using a statement like "`clk = ~clk`" would depend on the proper initialization of the clock signal to a value different than the default values of 1'bx or 'U'. Assigning explicit values also provides better control over the initial phase of the clock; you control whether the clock is starting high or low.

Sample 5-1. Generating a 50% duty-cycle clock

```
reg clk;
parameter cycle = 10;  // 100MHz clock
always
begin
   #(cycle/2) ;
   clk = 1'b0;
   #(cycle/2) ;
   clk = 1'b1;
end
```

Any deterministic waveform is easy to generate.

Waveforms with deterministic edge-to-edge relationships with an easily identifiable period are also easy to generate. It is a simple process of generating each edge in sequence, at the appropriate time. For example, Figure 5-2 outlines an apparently complex

waveform. However, Sample 5-2 shows that it is simple to generate.

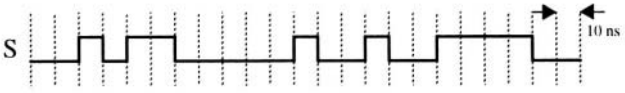


Figure 5-2. Apparently complex waveform

Sample 5-2. Generating a deterministic waveform

```
process
begin
   S <= '0'; wait for 20 ns;
   S <= '1'; wait for 10 ns;
   S <= '0'; wait for 10 ns;
   S <= '1'; wait for 20 ns;
   S <= '0'; wait for 50 ns;
   S <= '1'; wait for 10 ns;
   S <= '0'; wait for 20 ns;
   S <= '1'; wait for 10 ns;
   S <= '0'; wait for 20 ns;
   S <= '1'; wait for 40 ns;
   S <= '0'; wait for 20 ns;
   ...
end process;
```

The Verilog timescale may affect the timing of edges.

When generating waveforms in Verilog, you must select the appropriate timescale and precision to properly place the edges at the correct offset in time. When using an expression, such as "`cycle/2`", to compute delays, you must make sure that integer operations do not truncate a fractional part.

For example, the clock generated in Sample 5-3 produces a period of 14 ns because of truncation. If the precision is not sufficient, the delay values are rounded up or down, creating jitter on the edge location. For example, the clock generated in Sample 5-4 produces a period of 16 ns because of rounding. Only the signal generated in Sample 5-5 produces a 50 percent duty-cycle clock signal with a precise 15 ns period because the timescale offers the necessary precision for a 7.5 ns half-period.

Sample 5-3. Truncation errors in stimulus generation

```
`timescale 1ns/1ns
module testbench;
...
reg clk;
parameter cycle = 15;
always
begin
   #(cycle/2);  // Integer division
   clk = 1'b0;
   #(cycle/2);  // Integer division
   clk = 1'b1;
end
endmodule
```

Sample 5-4. Rounding errors in stimulus generation

```
`timescale 1ns/1ns
module testbench;
...
reg clk;
parameter cycle = 15;
always
begin
   #(cycle/2.0);   // Real division
   clk = 1'b0;
   #(cycle/2.0);   // Real division
   clk = 1'b1;
end
endmodule
```

Sample 5-5. Proper precision in stimulus generation

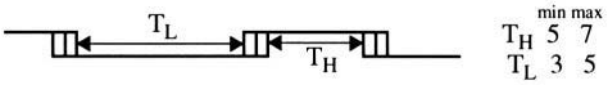
```
`timescale 1ns/100ps
module testbench;

reg clk;
parameter cycle = 15;
always
begin
   #(cycle/2.0);
   clk = 1'b0;
   #(cycle/2.0);
   clk = 1'b1;
end
endmodule
```

Generating a Complex Waveform

Avoid generating only a subset of possible complex waveforms.

A more complex waveform, with variations in the edge-to-edge timing relationships, requires more effort to model properly. Care must be taken not to overconstrain the waveform generation or to limit it to a subset of its possible variations. For example, if you generate the waveform illustrated in Figure 5-3 using the code in Sample 5-6, you generate only one of the many possible waveforms that meet the specification.

Figure 5-3. Complex waveform



Sample 5-6. Improperly generating a complex waveform

```
process
begin
   S <= '0'; wait for (5 ns + 7 ns) / 2;
   S <= '1'; wait for (3 ns + 5 ns) / 2;
end process;
```

Use a random number generator to model uncertainty.

To properly generate the complex waveform as specified, it is necessary to model the uncertainty of the edge locations within the minimum and maximum delay range. This can be easily accomplished by randomly generating a delay within the valid range. Verilog has a built-in system task to generate 32-bit random values called *$random.* VHDL does not have a built-in random function, but public-domain packages of varying complexity are available.[1] The code in Sample 5-7 properly generates a non-deterministic complex waveform.

Sample 5-7. Properly generating a complex waveform

```
process
begin
   S <= '0';
   wait for 5 ns + 2 ns * rnd_pkg.random;
   S <= '1';
   wait for 3 ns + 2 ns * rnd_pkg.random;
end process;
```

1. References to random number generation and linear-feedback shift register packages can be found in the *resources* section of:

   `http://janick.bergeron.com/wtb`

A linear random distribution may not yield enough interesting values.

To generate waveforms that are likely to stress the design under verification, it may be necessary to make sure that there are many instances of absolute minimum and absolute maximum values. With the linear random distribution produced by common random number generators, this is almost impossible to guarantee. You have to modify the waveform generator to issue more edges at the extremes of the valid range than would otherwise be produced by a purely linear random delay generation. In Sample 5-8, a function is used to skew the random distribution with 30 percent minimum value, 30 percent maximum value, and 40 percent random linear distribution within the valid range.

Sample 5-8. Skewing the linear random distribution

```
process
   function skewed_dist return real is
      variable distribute: real;
   begin
      distribute := rnd_pkg.random;
      if distribute < 0.3 then
         return 0.0;
      elsif distribute < 0.6 then
         return 1.0;
      else
         return rnd_pkg.random;
      end if;
   end skewed_dist;
begin
   S <= '0';
   wait for 5 ns + 2 ns * skewed_dist;
   S <= '1';
   wait for 3 ns + 2 ns * skewed_dist;
end process;
```

Generating Synchronized Waveforms

Most waveforms are not independent.

Stimuli for a design are never composed of a single signal. Multiple signals must be properly generated with respect to each other to properly stimulate the design under verification. When generating interrelated waveforms, you must be careful not to create race conditions and to properly align edges both in real time and in delta time.

Synchronized waveforms must be properly modeled.

The first signal to be generated after the clock signal is the hardware reset signal. These two signals must be properly synchronized to correctly reset the design. The generation of the reset signal should also reflect its synchronization with the clock signal. For

example, consider the specification for a reset signal shown in Figure 5-4. The code in Sample 5-9 shows how such a waveform is frequently generated.

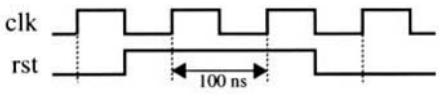


Figure 5-4. Reset waveform specification

Sample 5-9. Improperly generating a synchronized waveform

```
always
begin
   #50 clk = 1'b0;
   #50 clk = 1'b1;
end

initial
begin
   rst = 1'b0;
   #150 rst = 1'b1;
   #200 rst = 1'b0;
end
```

Race conditions can be easily created between synchronized signals.

There are two problems with the way these two waveforms are generated in Sample 5-9. The first problem is functional: there is a race condition between the *clk* and *rst* signals.[2] At simulation time 150, and again later at simulation time 350, both registers are assigned at the same timestep. Because the *blocking* assignment is used for both assignments, one of them is assigned first. A block sensitive to the falling edge of *clk* may execute before or after *rst* is assigned. From the perspective of that block, the specification shown in Figure 5-4 could appear to be violated. The race condition can be eliminated by using *non-blocking* assignments, as shown in Sample 5-10. Both *clk* and *rst* signals are assigned between timesteps, when no blocks are executing. If the design under verification uses the falling edge of *clk* as the active edge, *rst* is already - and reliably - assigned.

2. I did not bring up race conditions in the section titled "Read/Write Race Conditions" on page 141 just to conveniently forget about them here. Just to keep you on your toes, you'll see them appear throughout this book.

Sample 5-10. Race-free generation of a synchronized waveform

```
always
begin
   #50 clk <= 1'b0;
   #50 clk <= 1'b1;
end

initial
begin
   rst = 1'b0;
   #150 rst <= 1'b1;
   #200 rst <= 1'b0;
end
```

Lack of maintainability can introduce functional errors.

The second problem, which is just as serious as the first one, is maintainability of the description. You could argue that the first problem is more serious, since it is functional. The entire simulation can produce the wrong result under certain conditions. Maintainability has no such functional impact. Or has it? What if you made a change as simple as changing the phase or frequency of the clock. How would you know to also change the generation of the reset signal to match the new clock waveform?

Conditions in real life are different than within the confines of this book.

In the context of Sample 5-10, with Figure 5-4 nearby, you would probably adjust the generation of the *rst* signal. But outside this book, in the real world, these two blocks could be separated by hundreds of lines, or even be in different files. The specification is usually a document one inch thick, printed on both sides. The timing diagram shown in Figure 5-4 could be buried in an anonymous appendix, while the pressing requirements of changing the clock frequency or phase was urgently stated in an email message. And you were busy debugging this other testbench when you received that pesky email message! Would you know to change the generation of the reset signal as well? I know I would not.

Model the synchronization within the generation.

Waiting for an apparently arbitrary delay can move out-of-sync with respect to the delay of the clock generation. A much better way of modeling synchronized waveforms is to include the synchronization in the generation of the dependent signals, as shown in Sample 5-11. The proper way to synchronize the *rst* signal with the *clk* signal is for the generator to wait for the significant synchronizing event, whenever it may occur. The timing or phase of the clock generator can now be modified, without affecting the proper generation of the *rst* waveform. From the perspective of a design, sensi-

tive to the falling edge of *clk, rst* is reliably assigned one delta-cycle after the clock edge.

Sample 5-11. Proper generation of a synchronized waveform

```
always
begin
   #50 clk <= 1'b0;
   #50 clk <= 1'b1;
end

initial
begin
   rst = 1'b0;
   wait (clk !== 1'bx);
   @ (negedge clk);
   rst <= 1'b1;
   @ (negedge clk);
   @ (negedge clk);
   rst <= 1'b0;
end
```

Synchronized waveforms may be generated from a single block.

The maintainability argument can be taken one step further. Remember the section "Parallel vs. Sequential" on page 132? The sequence of the *rst* and *clk* waveforms is deterministic and can be modeled using a single sequential statement block, as shown in Sample 5-12. The synchronized portion of the *rst* and *clk* waveforms is generated first, then the remaining free-running *clk* waveform is generated. This generator differs from the one in Sample 5-11: from the perspective of a design sensitive to the falling edge of *clk, rst* has already been assigned.

Sample 5-12. Sequential generation of a synchronized waveform

```
initial
begin
   // Apply reset for first 2 clock cycles
   rst = 1'b0;
   #50 clk <= 1'b0;
   repeat (2) #50 clk <= ~clk;
   rst <= 1'b1;
   repeat (4) #50 clk <= ~clk;
   rst <= 1'b0;

   // Generate only the clock afterward
   forever #50 clk <= ~clk;
end
```

Delta delays are functionally equivalent to real delays.

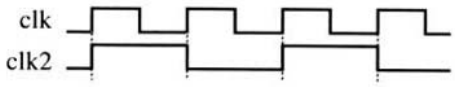
In the specification shown in Figure 5-4, the transition of *rst* is aligned with a transition on *clk*. The various ways of generating these two signals determined the sequence of these transitions, whether they occurred at the same delta cycle, in different delta cycles, or if their ordering was deterministic. Although delta-cycle delays are considered zero-delays by the simulator, functionally they have the same effect as real delays.

The next two sections, "Aligning Waveforms in Delta-Time" and "Generating Synchronous Data Waveforms" discuss how signals can be properly aligned or delayed to prevent unintentional functional delays.

Aligning Waveforms in Delta-Time

Derived waveforms, such as the one shown in Figure 5-5, are apparently easy to generate. A simple process, sensitive to the proper edge of the original signal as shown in Sample 5-13, and voila! Even the waveform viewer shows that it is right!

Figure 5-5. Derived waveform specification



Sample 5-13. Improperly generating a derived waveform

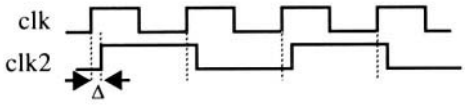
```
clk2_gen: process(clk)
begin
   if clk = '1' then
      clk2 <= not clk2;
   end if;
end process clk2_gen;
```

Watch for delta delays in derived waveforms.

The problem is not visually apparent. Because of the simulation cycle (See "The Simulation Cycle" on page 129), there is a delta cycle between the rising edge of the base clock signal, and the transition on the derived clock signal, as shown in Figure 5-6. Any data transferred from the base clock domain to the derived clock domain goes through this additional delta cycle delay. In a zero-delay simulation, such as a behavioral or RTL model, this additional delta-cycle delay can have the same effect as an entire clock cycle delay.

To maintain the relationship between the base and derived signals, their respective edges must be aligned in delta time. The only way to perform this task is to re-derive the base signal, as shown in Sample 5-14 and illustrated in Figure 5-7. The base signal is never used by other processes. Instead, they must use the derived base signal.

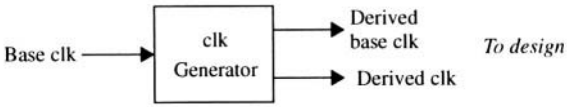
Figure 5-6. Delta delay in derived waveform



Sample 5-14. Properly generating a derived waveform

```
derived_gen: process(clk)
begin
   clk1 <= clk;
   if clk = '1' then
      clk2 <= not clk2;
   end if;
end process derived_gen;
```

Figure 5-7. Generation of aligned derived signals



Generating Synchronous Data Waveforms

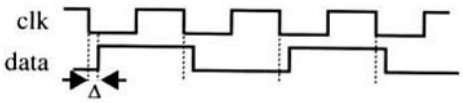
There is a race condition between the clock and data signal.

Sample 5-10, Sample 5-11, and Sample 5-15 show how you could generate a zero-delay synchronous data waveform. In Sample 5-11 and Sample 5-15, it is identical to the way flip-flops are inferred in an RTL model. As illustrated in Figure 5-8, there is a delay between the edge on the clock and the transition on *data,* but it is a single delta cycle. In terms of simulation time, there is no delay. For RTL models, this infinitesimal clock-to-Q delay is sufficient to properly model the behavior of synchronous circuits. However, this assumes that all clock edges are aligned in delta time (see "Aligning Waveforms in Delta-Time" on page 164). If you are generating both clock and data signals from the outside of the model of the design under verification, you have no way of ensuring that the total number of delta-cycle delays between the clock and the data is maintained, or at least be in favor of the data signal!

Sample 5-15. Zero-delay generation of synchronous data

```
sync_data_gen: process(clk)
begin
   if clk = '0' then
      data <= ...;
   end if;
end process sync_data_gen;
```

Figure 5-8. Synchronous data waveforms

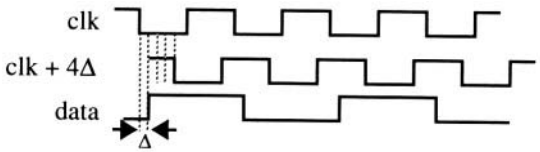


The clock may be delayed more than the data.

For many possible reasons, the clock signal may be delayed by more delta cycles than its corresponding data signal. These delays could be introduced by using different I/O pad models for the clock and data pins. They could also be introduced by the clock distribution network, which does not exist on the data signal. If the clock signal is delayed more than the data signal, even in zero-time as shown in Figure 5-9, the effect is the same as removing an entire clock cycle from the data path.

Interface specifications never specify zero-delay values. A physical interface always has a real delay between the active edge of a clock signal and its synchronous data. When generating synchronous data, always provide a real delay between the active edge and the transition on the data signal, as shown in Sample 5-16 and Sample 5-17.

Figure 5-9. Delta delays in clock path



Sample 5-16. Non-zero-delay generation of synchronous data

```
sync_data_gen: process(clk)
begin
   if clk = '0' then
      data <= ... after 1 ns;
   end if;
end process sync_data_gen;
```

Sample 5-17. Sequential generation of a delayed synchronized data waveform

```
initial
begin
   // Apply reset for first 2 clock cycles
   rst = 1'b0;
   #50 clk <= 1'b0;
   repeat (2) #50 clk <= ~clk;
   rst <= #1 1'b1;
   repeat (4) #50 clk <= ~clk;
   rst <= #1 1'b0;

   // Generate only the clock afterward
   forever #50 clk <= ~clk;
end
```

Encapsulating Waveform Generation

The generation of waveforms may need to be repeated during a simulation.

There is a problem with the way the *rst* waveform is generated in Sample 5-17. What if it were necessary to reset the device under verification multiple times during the execution of a testbench? One example would be to execute multiple testcases in a single simulation. Another one is the "hardware reset" testcase which verifies that the reset operates properly. In that respect, the code in Sample 5-11 is closer to an appropriate solution. The only thing that needs to be changed is the use of the *initial* block. The *initial* block runs only once and is eliminated from the simulation once completed. There is no way to have it execute again during a simulation.

Encapsulate waveform generation in a subprogram.

The proper mechanism to encapsulate statements that you may need to repeat during a simulation is to use a *task* or a *procedure* as shown in Sample 5-18. To repeat the waveform, simply call the subprogram. To maintain the behavior of using an *initial* block to automatically reset the device under verification at the beginning of the simulation, simply call the task in an *initial* block. Pop quiz: what is missing from the *hw_reset* task in Sample 5-18? The answer can be found in this footnote.[3]

A subprogram can be used to properly apply vectors.

Another example of a synchronized waveform whose generation can be encapsulated is the application of input vector data. As illus-

3. The task *hw_reset* contains delay control statements. It should contain a semaphore to detect concurrent activation. You can read more about this issue in "Non-Reentrant Tasks" on page 151.

Sample 5-18. Encapsulating the generation of a synchronized waveform

```
always
begin
   #50 clk <= 1'b0;
   #50 clk <= 1'b1;
end

task hw_reset;
begin
   rst = 1'b0;
   wait (clk !== 1'bx);
   @ (negedge clk);
   rst <= 1'b1;
   @ (negedge clk);
   @ (negedge clk);
   rst <= 1'b0;
end
endtask
initial hw_reset;
```

trated in Figure 5-10, vector data must be applied with a proper setup and hold time - but no more - to meet the input timing constraints. Instead of repeating the synchronization for each vector, a subprogram can be used for synchronization with the input clock. It would also apply the vector data received as input argument. The code in Sample 5-19 shows the implementation and use of a *task* applying input vectors according to the specification in Figure 5-10. Notice how the input is set to unknowns after the specified hold time to stress the timing of the interface. Leaving the input to a constant value would not detect cases where the device under verification does not meet the maximum hold requirement.

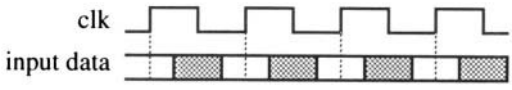


Figure 5-10. Input data waveform specification

Sample 5-19. Encapsulating the application of input data vectors

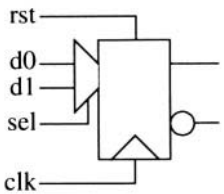
```
task apply_vector;
   input [...] vector;
begin
   inputs <= vector;
   @(posedge clk);
   #(Thold);
   inputs <= ...'bx;
   #(cycle - Thold - Tsetup);
end
endtask

initial
begin
   hw_reset;
   apply_vector(...);
   apply_vector(...);
   ...
end
```

Abstracting Waveform Generation

Vectors are difficult to write and maintain.

Using synchronous test vectors to verify a design is rather cumbersome. They are hard to interpret and difficult to correctly specify. For example, using vectors to verify a synchronously resetable D flip-flop with a 2-to-1 multiplexer on the input, as shown in Figure 5-11, could be stimulated using the vectors shown in Sample 5-20.

Figure 5-11. 2-to-1 input sync reset D flip-flop.



Use subprograms to encapsulate operations.

It would be easier if the operation accomplished by the vectors were abstracted. The device under verification can only perform three things:

- A synchronous reset
- Load from input *d0*
- Load from input *d1*

Sample 5-20. Test vectors for 2-to-l input sync reset D flip-flop

```
initial
begin
   // Vector: rst, d0, d1, sel
   apply_vector(4'b1110);
   apply_vector(4'b0100);
   apply_vector(4'b1111);
   apply_vector(4'b0011);
   apply_vector(4'b0010);
   apply_vector(4'b0011);
   apply_vector(4'b1111);
   ...
end
```

Instead of providing vectors to repeatedly perform these operations, why not provide subprograms that perform these operations? All that will be left is to call the subprograms in the appropriate order, with the appropriate data.

Try to apply the worst possible combination of inputs.

The subprogram to perform the synchronous reset is very simple. It needs to assert the *rst* input, then wait for the active edge of the clock. But what about the other inputs? You could decide to leave them unchanged, but is that the worst possible case? What if the reset was not functional and the device loaded one of the inputs and that input was set to '0' ? It would be impossible to differentiate the wrong behavior from the correct one. To create the worst possible condition, both *d0* and *d1* inputs must be set to '1'. The *sel* input can be set randomly, since either input selection should be functionally identical. An implementation of the *sync_reset* task is shown in Sample 5-21.

Sample 5-21. Abstracting the sync reset operation

```
task sync_reset;
begin
   rst <= 1'b1;
   d0   <= 1'b1;
   d1   <= 1'b1;
   sel <= $random;
   @  (posedge clk);
   #(Thold);
   {rst, d0, d1, sel} <= 4'bxxxx;
   #(cycle - Thold - Tsetup);
end
endtask
```

Pass input values as arguments to the subprogram.

The second operation this design can perform is to load input *d0*. The *task* to perform this operation is shown in Sample 5-22. Unlike resetting the design, loading data can have different input values: it can load either a '1' or a '0'. The value of the input to load is passed as an argument to the task. The worst condition is created when the other input is set to the complement of the input value on *d0*. If the device is not functioning properly and is loading from the wrong input, then the result will be clearly wrong.

Sample 5-22. Abstracting the load *d0* operation

```
task load_d0;
   input data;
begin
   rst <= 1'b0;
   d0  <= data;
   d1  <= ~data;
   sel <= 1'b0;
   @ (posedge clk);
   #(Thold);
   {rst, d0, d1, sel} <= 4'bxxxx;
   #(cycle - Thold - Tsetup);
end
endtask
```

Stimulus generated with abstracted operations is easier to write and maintain.

The last operation this design can perform is to load input *d1*. The *task* abstracting the operation to load from input *d1* is similar to the one shown in Sample 5-22. Once operation abstractions are available, providing the proper stimulus to the design under verification is easy to write and understand. Compare the code in Sample 5-23 with the code of Sample 5-20. If the polarity of the *rst* input were changed, which verification approach would be easiest to modify?

Sample 5-23. Verifying the design using operation abstractions

```
initial
begin
   sync_reset;
   load_d0(1'b1);
   sync_reset;
   load_d1(1'b1);
   load_d0(1'b0);
   load_d1(1'b1);
   sync_reset;
   ...
end
```

VERIFYING THE OUTPUT

Generating stimulus is only half of the job. Actually, it is more like 30 percent of the job. The other part, verifying that the output is as expected, is much more time-consuming and error-prone. There are various ways the output can be checked against expectations. They have varying degrees of applicability and repeatability.

Visual Inspection of Response

Results can be printed.

The most obvious method for verifying the output of a simulation is to visually inspect the results. The visual display can be an ASCII printout of the input and output values at specific points in time, as shown in Sample 5-24.

Sample 5-24. ASCII view of simulation results

```
      r  s
      sddeqq
Time  t011 b
----------------
0100  1110xx
0105  111001
0200  010001
0205  010010
0300  111110
0305  111101
0400  001101
0405  001110
0500  001010
0505  001010
0600  001110
0605  001110
0700  111110
0705  111101
...
```

Producing Simulation Results

To print simulation results, you must model the signal sampling.

The specific points in time that are significant for a particular design or testbench are always different. Which signals are significant is also different and may change as the simulation progresses. If you know which time points and signals are significant for determining the correctness of the simulation results, you have to be able to model that knowledge. Producing the proper simulation results involves modeling the behavior of the signal sampling.

Many sampling techniques can be used.

There are many sampling techniques, each as valid as the other. The correct sampling technique depends on your needs and on what makes the simulation results significant. Just as you have to decide which input sequence is relevant for the functionality you are trying to verify, you must also decide on the output sampling that is relevant for determining the success or failure of the function under verification.

You can sample at regular intervals.

The simplest sampling technique is to sample the relevant signals at a regular interval. The interval can be an absolute delay value, as illustrated in Sample 5-25, or a reference signal such as the clock, as illustrated in Sample 5-26.

Sample 5-25. Sampling at a delay interval

```
parameter INTERVAL = 10;
always
begin
   #(INTERVAL);
   $write(...);
end
```

Sample 5-26. Sampling based on a reference signal

```
process (clk)
   variable L: line;
begin
   if clk'event and clk = '0' then
      write(L, ...);
      writeline(output, L);
   end if;
end process;
```

You can sample based on a signal changing value.

Another popular sampling technique is to sample a set of signals whenever one of them changes. This is used to reduce the amount of data produced during a simulation when signals do not change at a constant interval.

To sample a set of signals, simply make a *process* or *always* block sensitive to the signals whose changes are significant, as shown in Sample 5-27. The set of signals displayed and monitored can be different. Verilog has a built-in task, called *$monitor,* to perform this sampling when the set of display and monitored signals are identical.

An example of using the *$monitor* task is shown in Sample 5-28. Its behavior is different from the VHDL sampling process shown in Sample 5-27: changes in values of signals *rst, d0, d1, sel, q,* and *qb*

cause the display of simulation results, whereas only changes in *q* and *qb* trigger the sampling in the VHDL example. Note that Verilog simulations are limited to a single active *$monitor* task. Any subsequent call to *$monitor* replaces the previous monitor.

Sample 5-27. Sampling based on signal changes

```
process (q, qb)
   variable L: line;
begin
   write(L, rst & d0 & d1 & sel & q & qb) ;
   writeline(output, L) ;
end process;
```

Sample 5-28. Sampling using the *$monitor* task

```
initial
begin
   $monitor("...", rst, d0, d1, sel, q, qb);
end
```

Minimizing Sampling

To improve simulation performance, minimize sampling.

The use of an output device on a computer slows down the execution of any program. Therefore, the production of simulation output reduces the performance of the simulation. To maximize the speed of a simulation, minimize the amount of simulation output produced during its execution.

In Verilog, an active *$monitor* task can be turned on and off by using the *$monitoron* and *$monitoroff* tasks, respectively. If you are using an explicit sampling *always* block or are using VHDL, you should include sampling minimization techniques in your model, as illustrated in Sample 5-29. A very efficient way of minimizing sampling is to have the stimulus turn on the sampling when an interesting section of the testcase is entered, as shown in Sample 5-30.

Visual Inspection of Waveforms

Results are better viewed when plotted over time.

Waveform displays usually provide a more intuitive visual representation of simulation results. Figure 5-12 shows the same information as Sample 5-24, but using a waveform view. The waveform view has the advantage of providing a continuous display of many values over the entire simulation time, not just at specific time points as in a text view. Therefore, you need not specify or model a particular sampling technique. The signals are continuously sam-

Sample 5-29. Minimizing sampling

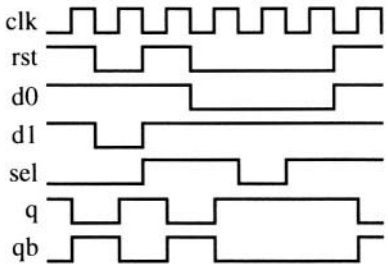
```
process
begin
   wait until <interesting_condition>;
   sampling: loop
      wait on q, qb;
      write(l, rst & d0 & d1 & sel & q & qb);
      writeline(output, l);
      exit sampling
         when not <interesting_condition>;
   end loop sampling;
end process;
```

Sample 5-30. Controlling the sampling from the stimulus

```
initial
begin
   $monitor("...", rst, d0, d1, sel, q, qb);
   $monitoroff;
   sync_reset;
   load_d0(1'b1);
   sync_reset;
   $monitoron;
   load_d1(1'b1);
   load_d0(1'b0);
   load_d1(1'b1);
   sync_reset;
   $monitoroff;
   ...
end
```

pled, usually into an efficient database format. Sampling for waveforms must be turned on explicitly. It is a tool-dependent process that is different for each language and each tool.

Figure 5-12. Waveform view of simulation results



Minimize the number and duration of sampled signals.

The default behavior is to sample all signals during the entire simulation. The waveform sampling process consumes a significant portion of the simulation resources. Reducing the number of signals

sampled, or the duration of the sampling, increases the simulation performance.

SELF-CHECKING TESTBENCHES

This section introduces a reliable and reproduceable technique for output verification: testbenches that verify themselves. I discuss the pros and cons of popular vector-based implementation techniques. I show how to verify the simulation results at run-time by modelling the expected response at the same time as the stimulus.

Visual inspection is not acceptable.

The model of the D flip-flop with a 2-to-1 input mux being verified has a functional error. Can you identify it using either views of the simulation results in Sample 5-24 or Figure 5-12? How long did it take to diagnose the problem?[4]

This example was for a very simple design, over a very short period of time, and for a very small number of signals. Imagine visually inspecting simulation results spanning hundreds of thousands of clock cycles, and involving hundreds of input and output signals. Then imagine repeating this visual inspection for every testbench, and every simulation of every testbench. The probability that you will miss identifying an error is equal to one. You must automate the process of comparing the simulation results against the expected outputs.

Input and Output Vectors

Specify the expected output values for each clock cycle.

The first step in automating output verification is to include the expected output with the input stimulus for every clock cycle. The vector application task in Sample 5-19 can be easily modified to include the comparison of the output signals with the specified output vector, as shown in Sample 5-31. The testcase becomes a series of input/output test vectors, as shown in Sample 5-32.

Test vectors require synchronous interfaces.

The main problem with input and output test vectors (other than the fact that they are very difficult to specify, maintain, and debug), is that they require perfectly synchronous interfaces. If the design under verification contains interfaces in different clock domains,

4. The logic value on input *d0* is ignored and a '1' is always loaded.

Sample 5-31. Application of input and verification of output data vectors

```
task apply_vector;
   input [ . . . ] in_data;
   input [...] out_data;
begin
   inputs <= in_data;
   @(posedge clk);
   fork
      begin
         #(Thold);
         inputs <= ...'bx;
      end
      begin
         #(Td);
         if (outputs !== out_data) ...;
      end
      #(cycle - Thold - Tsetup);
   join
end
endtask
```

Sample 5-32. Input/output test vectors for 2-to-l input sync reset D flip-flop

```
initial
begin
   // In: rst, d0, d1, sel
   // Out: q, qb
   apply_vector(4'b1110, 2'b00)
   apply_vector(4'b0100, 2'b10)
   apply_vector(4'b1111, 2'b00)
   apply_vector(4'b0011, 2'b10)
   apply_vector(4'b0010, 2'b01)
   apply_vector(4'b0011, 2'b10)
   apply_vector(4'b1111, 2'b00)
   ...
end
```

each requires its own test vector stream. If any interface contains asynchronous signals, they have to be either externally synchronized before vectors are applied, or treated as synchronous signals, therefore under-constraining the verification.

Golden Vectors

A set of reference simulation results can be used.

The next step toward automation of the output verification is the use of golden vectors. It is a simple extension of the manufacturing test process where devices are physically subjected to a series of qualifying test vectors. A set of reference output results, determined

to be correct, are kept in a file or database. The simulation outputs are captured in a similar format during a simulation. They are then compared against the reference results. Golden vectors have an advantage over input/output vectors because the expected output values need not be specified in advance.

Text files can be compared using *diff*.

If the simulation results are kept in ASCII files, the simplest comparison process involves using the UNIX *diff* utility. The *diff* output for the simulation results shown in Sample 5-24 is shown in Sample 5-33. You can appreciate how difficult the subsequent task of diagnosing the functional error will be.

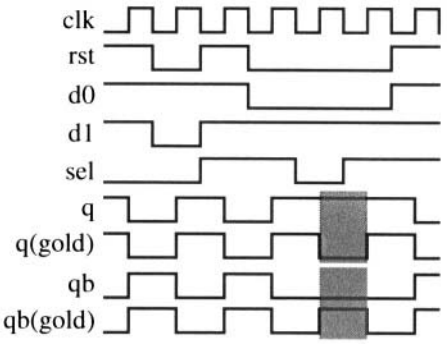
Sample 5-33. *diff* output of comparing ASCII view of simulation results

```
14c2
>0505 001010
>0600 001110
--------
<0505 001001
<0600 001110
...
```

Waveforms can be compared by a specialized tool.

Waveform comparators can also be used. They are tools similar to waveform viewers and are usually built into one. They compare two sets of waveforms then highlight the differences on a graphical display. The display of a waveform comparator might look something like the results illustrated in Figure 5-13. Identifying the problem is easier since you have access to the entire history of the simulation in a single view.

Figure 5-13. Waveform differences in simulation results



Golden vectors must still be visually inspected.

The main problem with golden simulation results is that they need to be visually inspected to be determined as valid. This self-checking technique only reduces the number of times a set of simulation responses must be visually verified, not the need for visual inspection. The result from each testbench must still be manually confirmed as good.

Golden vectors do not adapt to changes.

Another problem: reference simulation results do not adapt to modifications in the design under verification that may only affect the timing of the result, without affecting its functional correctness. For example, an extra register may be added in the datapath of a design to help meet timing constraints. All that was added was a pipeline delay. The functionality was not modified. Only the latency was increased. If that latency is irrelevant to the functional correctness of the overall system, the reference vectors must be updated to reflect that change.

Golden vectors require a significant maintenance effort.

Reference simulation results must be visually inspected for every testcase, and modified or regenerated whenever a change is made to the design, each time requiring visual inspection. Using reference vectors is a high-maintenance, low-efficiency self-checking strategy. Verification vectors should be used only when a design must be 100 percent backward compatible with an existing device, signal for signal, clock cycle for clock cycle. In those circumstances, the reference vectors never change and never require visual inspection as they are golden by definition.

Separate the reference vectors along clock domains.

Reference simulation results also work best with synchronous interfaces. If you have multiple interfaces in separate clock domains, it is necessary to generate reference results for each domain in a separate file. If a single file is used, the asynchronous relationship between the clock domains may result in the samples from different domains being written in a different order. The ordering difference is not functionally relevant, but would be flagged as an error by the comparison tool.

Run-Time Result Verification

Comparing simulation results against a reference set of vectors or waveforms is a post-processing process. It is possible to verify the correctness of the simulation results at runtime, in parallel with the stimulus generation.

You can use a reference model.

Using a reference model is a simple extension of the golden vector technique. As illustrated in Figure 5-14, the reference model and the design under verification are subjected to the same stimulus and their output is constantly monitored and compared for discrepancies.

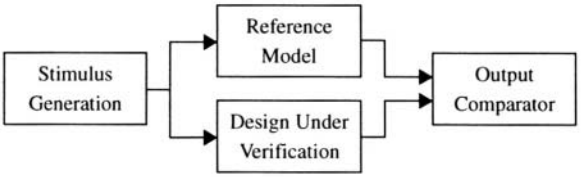


Figure 5-14. Using a reference model

In reality, a reference model never works.

The reality of reference models is different. They rarely exist. When they do, they are in a language that cannot be easily integrated with either VHDL or Verilog. When they can be integrated, they produce output with a different timing or accuracy, making the output comparison impractical. When all of these obstacles are overcome, they are often a burden on the simulation performance. Using reference simulation results, as described in the previous section, is probably a better alternative.

You can model the expected response.

If you know what you are looking for when visually inspecting simulation results, you should be able to describe it also. It should be part of the testcase specification. If the expected response can be described, it can be modeled. If it can be modeled, it can be included in the testbench. By including the expected response in the testbench, it is able to determine automatically whether the testcase succeeded or failed.

Focus on operations instead of input and output vectors.

In "Abstracting Waveform Generation" on page 169, subprograms were used to apply stimulus to the design. These subprograms abstracted the vectors into atomic operations that could be performed on the design. Why not include the verification of the operation's output as part of the subprogram? Instead of simply applying inputs, then leaving the output verification to a separate process, integrate both the stimulus and response checking into complete operations. Performing the verification becomes a matter of verifying that operations, individually and in sequence, are performed appropriately.

For example, the task shown in Sample 5-21 can include the verification that the flip-flop was properly reset as shown in Sample 5-

34. Similarly, the task used to apply the stimulus to load data from the *d0* input shown in Sample 5-22 can be modified to include the verification of the output, as shown in Sample 5-35. The testcase shown in Sample 5-23 now becomes entirely self-checking.

Sample 5-34. Verifying the sync reset operation

```
task sync_reset;
begin
   rst <= 1'b1;
   d0  <= 1'b1;
   d1  <= 1'b1;
   sel <= $random;
   @ (posedge clk);
   #(Thold);
   if (q !== 1'b0 || qb !== 1'b1) ...
   {rst, d0, d1, sel} <= 4'bxxxx;
   #(cycle - Thold - Tsetup);
end
endtask
```

Sample 5-35. Verifying the load *d0* operation

```
task load_d0;
   input data;
begin
   rst <= 1'b0;
   d0  <= data;
   d1  <= ~data;
   sel <= 1'b0;
   @ (posedge clk);
   #(Thold);
   if (q !== data || qb !== ~data) ...
   {rst, d0, d1, sel} <= 4'bxxxx;
   #(cycle - Thold - Tsetup);
end
endtask
```

Make sure the output is properly verified.

The problem with output verification is that you can't identify a functional discrepancy if you are not looking at it. Using an *if* statement to verify the output in the middle of a stimulus process only looks at the output value for a brief instant. That may be acceptable, but it does not say anything about the *stability* of that output. For example, the tasks in Sample 5-34 and Sample 5-35 only check the value of the output at a single point. Figure 5-15 shows the complete specification for the flip-flop. The verification sampling point is shown as well.

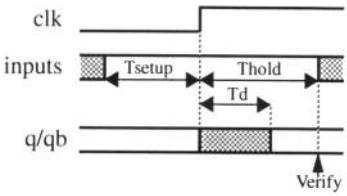


Figure 5-15. Timing specification for the flip-flop

Make sure you verify the output over the entire significant time period.

To properly and completely verify the functionality of the design, it is necessary to verify that the output is stable, except for the short period after the rising edge of the clock. That could be easily verified using a static timing analysis tool and a set of suitable constraints to verify against. If you want to perform the verification in Verilog or VHDL, the stability of the output cannot be easily verified in the same subprogram that applies the input. The input follows a deterministic data and timing sequence, whereas monitoring stability requires that the testbench code be ready to react to any unexpected changes. Instead, it is better to use a separate monitor process, executing in parallel with the stimulus. The stimulus subprogram can still check the value. The stability monitor, as shown in Sample 5-36, simply verifies that the output remains stable, whatever its value. In VHDL, the *'stable* attribute was designed for this type of application, as shown in Sample 5-37. The stability of the output signal can be verified in the stimulus procedure, but it requires prior knowledge of the clock period to perform the timing check.

Sample 5-36. Verifying the stability of flip-flop outputs

```
initial
begin
   // wait for the first clock edge
   @ (posedge clk);
   forever begin
      // Ignore changes for Td after clock edge
      #(Td);
      // Watch for a change before the next clk
      fork: stability_mon
         @ (q or qb) $write("...");
         @ (posedge clk) disable stability_mon;
      join
   end
end
```

Sample 5-37. Verifying the load *d0* operation and output stability

```
procedure load_d0(data: std_logic) is
begin
   rst <= '0';
   d0  <= data;
   d1  <= not data;
   sel <= '0';
   wait until clk = '1';
   assert q'stable(cycle - Td) and
          qb'stable(cycle - Td);
   wait for Thold;
   rst <= 'X';
   d0  <= 'X';
   d1  <= 'X';
   sel <= 'X' ;
   assert q = data and qb = not data;
   wait for cycle - Thold - Tsetup;
end load_d0;
```

COMPLEX STIMULUS

This section introduces more complex stimulus generation scenarios through the use of bus-functional models. I start with non-deterministic stimulus, where the stimulus or its timing depends on answers from the device under verification. I also show how to avoid wasting precious simulation cycles by getting caught in deadlock conditions. I explain how to generate asynchronous stimulus and more complex protocols such as CPU cycles. Finally, I show how to write configurable bus-functional models.

Generating inputs may require cooperating with the design.

Applying stimulus to a clock or reset input is straightforward. You are under complete control of the timing of the input signal. However, if the interface being driven contains handshaking or flow-control signals, the generation of the stimulus requires cooperation with the design under verification.

Feedback Between Stimulus and Design

Without feedback, verification can be under-constrained.

Figure 5-16 shows the specification for a simple bus arbiter. If you were to verify the design of the arbiter using test vectors applied at every clock cycle, as described in “Input and Output Vectors” on page 176, you would have to assume a specific delay between the assertion of the *req* signal and the assertion of the *grt* signal. Any delay value between one and five clock cycles would be functionally correct, but the only reliable choice is a delay of five cycles.

Similarly, a delay of three clock cycles would have to be made for the release portion of the verification. These choices, however, severely under-constrain the verification. If you want to stress the arbiter by issuing requests as fast as possible, you would want to know when the request was granted and released, so it could be reapplied as quickly as possible.

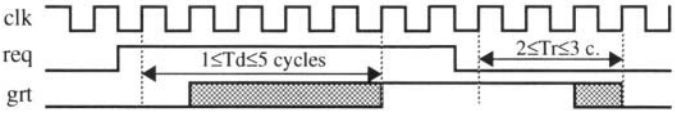


Figure 5-16. Specification for a simple arbiter

Stimulus generation can wait for feedback before proceeding.

If, instead of using input and output test vectors, you are using encapsulated operations to verify the design, you can modify the operation to wait for feedback from the design under verification before proceeding. You should also include any timing and functional verification in the feedback monitoring to ensure that the design responds in an appropriate manner. Sample 5-38 shows the *bus_request* operation procedure. It samples the *grt* signal at every clock cycle, and immediately returns once it detects that the bus was granted. With a similarly implemented *bus_release* procedure, a testcase that stresses the arbiter under maximum load can be easily written, as shown in Sample 5-39.

Sample 5-38. Verifying the bus request operation

```
procedure bus_request is
   variable cycle_count: integer := 0;
begin
   req <= '1';
   wait until clk = '1';
   while grt = '0' loop
      wait until clk = '1';
      cycle_count := cycle_count + 1;
   end loop;
   assert 1 <= cycle_count and cycle_count <= 5;
end bus_request;
```

Recovering from Deadlocks

A deadlock may prevent the testcase from running to completion.

There is a risk inherent to using feedback in generating stimulus: the stimulus now depends on the proper operation of the design under verification to complete. If the design does not provide the feedback as expected, the stimulus generation may be halted, waiting for a condition that will never occur. For example, consider the

Sample 5-39. Stressing the bus arbiter.

```
test_sequence: process
   procedure bus_request ...
   procedure bus_release ...
begin
   for I in 1 to 10 loop
      bus_request;
      bus_release;
   end loop;
   assert false severity failure;
end process test_sequence;
```

bus_request procedure in Sample 5-38. What happens if the *grt* signal is never asserted? The procedure remains stuck in the *while* loop and never returns.

A deadlocked simulation appears to be running correctly.

If this were to occur, the simulation would still be running, merrily going around and around the *while* loop. The simulation time would advance at each tick of the clock. The CPU usage of your workstation would show near 100 percent usage. The only symptom that something is wrong would be that no messages are produced on the simulation's output log and the simulation runs for much longer than usual. If you are watching the simulation run and expect regular messages to be produced during its execution, you would quickly recognize that something is wrong and manually interrupt it.

A deadlocked simulation wastes regression runs.

But what if there is no one watching the simulation, such as during a regression run? Regressions are large scale simulation runs where all available testcases are executed. They are used to verify that the functionality of the design under verification is still correct after modifications. Because of the large number of testcases involved in a regression, the process is automated to run unattended, usually overnight and on many computers. If a design modification creates a deadlock situation, all testcases scheduled to execute subsequently will never run, as the deadlocked testcase never terminates. The opportunity of detecting other problems in the regression run is wasted. It will be necessary to wait for another 24-hour period before knowing if the new version of the design meets its functional specification.

Eliminate the possibility of deadlock conditions.

When generating stimulus, you must make sure that there is no possibility of a deadlock condition. You must assume that the feedback condition you are waiting for may never occur. If the feedback condition fails to happen, you must then take appropriate action. It

could include terminating the testcase, or jumping to the next portion of the testcase that does not depend on the current operation, or attempting to repeat the operation after some delay. Sample 5-38 was modified as shown in Sample 5-40 to avoid the deadlock condition created if the arbiter failed and the *grt* signal was never asserted.

Sample 5-40. Avoiding deadlock in the bus request operation

```
procedure bus_request is
   variable cycle_count: integer := 0;
begin
   req <= '1';
   wait until clk = '1';
   while grt = '0' loop
      wait until clk = '1';
      cycle_count := cycle_count + 1;
      assert cycle_count < 500
         report "Arbiter is not working"
         severity failure;
   end loop;
   assert 1 <= cycle_count and cycle_count <= 5;
end bus_request;
```

Sample 5-41. Returning status in the bus request operation

```
procedure bus_request(good: out boolean) is
   variable cycle_count: integer := 0;
begin
   good := true;
   req <= '1';
   wait until clk = '1';
   while grt = '0' loop
      wait until clk = '1';
      cycle_count := cycle_count + 1;
      if cycle_count > 500 then
         good := false;
         return;
      end if;
   end loop;
   assert 1 <= cycle_count and cycle_count <= 5;
end bus_request;
```

Operation subprograms could return status.

If a failure of the feedback condition is detected, terminating the simulation on the spot, as shown in Sample 5-40, is easy to implement in each operation subprogram. If you want more flexibility in handling a non-fatal error, you might want to let the testcase handle the error recovery, instead of handling it inside the operation subprogram. The subprogram must provide an indication of the status of the operation's completion back to the testcase. Sample 5-41

shows the *bus_request* procedure that includes a *good* status flag indicating whether the bus was granted or not. The testcase is then free to attempt other bus request operations until it succeeds, as shown in Sample 5-42. Notice how the testcase takes care of avoiding its own deadlock condition if the bus request operation never succeeds.

Sample 5-42. Handling failures in the *bus_request* procedure

```
testcase: process
   variable granted : boolean;
   variable attempts: integer := 0;
begin
   ...
   attempts := 0;
   loop
      bus_request(granted);
      exit when granted;
      attempts := attempts + 1;
      assert attempts < 5
         report "Bus was never granted"
         severity failure;
   end loop;
   ...
end process testcase;
```

Asynchronous Interfaces

Test vectors underconstrain asynchronous interfaces.

Test vectors are inherently synchronous. The inputs are all applied at the same time. The outputs are all verified at the same time. And this process is repeated at regular intervals. Many interfaces, although implemented using finite state machines and edge-triggered flip-flops, are specified in an asynchronous fashion. The implementer has arbitrarily chosen a clock to streamline the physical implementation of the interface. If that clock is not part of the specification, it should not be part of the verification. For example, Figure 5-17 shows an asynchronous specification for a bus arbiter. Given a suitable clock frequency, the synchronous specification shown in Figure 5-16 would be a suitable implementation.

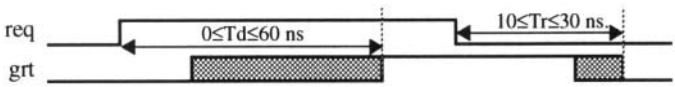


Figure 5-17. Asynchronous specification for a simple arbiter

Verify the synchronous implementation against the asynchronous specification.

Even though a clock may be present in the implementation, if it is not part of the specification, you cannot use it to generate stimulus nor to verify the response. You would be verifying against a particular implementation, not the specification. For example, a VME bus is asynchronous. The verification of a VME interface cannot make use of the clock used to implement that interface. If a clock is present, and the timing constraints make reference to clock edges, then you must use it to generate stimulus and verify response. For example, a PCI bus is synchronous. A verification of a PCI interface must use the PCI system clock to verify any implementation.

Behavioral code does not require a clock like RTL code.

Testbenches are written using behavioral code. Behavioral models do not require a clock. Clocks are artifices of the implementation methodology and are required only for RTL code. The bus request phase of the asynchronous interface specified in Figure 5-17 can be verified asynchronously with the *bus_request* procedure shown in Sample 5-43 or Sample 5-44. Notice how neither model of the bus request operation uses a clock for timing control. Also, notice how the Verilog version, in Sample 5-44, uses the definitely non-synthesizeable *fork/join* statement to wait for the rising edge of *grt* for a maximum of 60 time units.

Sample 5-43. Verifying the asynchronous bus request operation in VHDL

```
procedure bus_request(good: out boolean) is
begin
   req <= '1';
   wait until grt = '1' for 60 ns;
   good := grt = '1';
end bus_request;
```

Sample 5-44. Verifying the asynchronous bus request operation in Verilog

```
task bus_request;
   output good;
begin
   req = 1'b1;
   fork: wait_for_grt
      #60              disable wait_for_grt;
      @ (posedge grt)  disable wait_for_grt;
   join
   good = (grt == 1'b1);
end
endtask
```

Consider all possible failure modes.

There is one problem with the models of the bus request operation in Sample 5-43 and Sample 5-44. What if the arbiter was function-

ally incorrect and left the *grt* signal always asserted? Both models would never see a rising edge on the *grt* signal. They would eventually exhaust their maximum waiting period then detect *grt* as asserted, indicating a successful completion. To detect this possible failure mode, the bus request operation must verify that the *grt* signal is not asserted prior to asserting the *req* signal, as shown in Sample 5-45.

Sample 5-45. Verifying all failure modes in the asynchronous bus request operation

```
task bus_request;
   output good;
begin: bus_reguest_task
   if (grt == 1'b1) begin
      good = 1'b0;
      disable bus_request_task;
   end
   req = 1'b1;
   fork: wait_for_grt
      #60              disable wait_for_grt;
      @ (posedge grt)  disable wait_for_grt;
   join
   good = (grt == 1'b1);
end
endtask
```

Were you paying attention?

Pop quiz: The first *disable* statement in Sample 5-45 aborts the *bus_request* task and returns control to the calling block of the statement. Why does it disable the *begin/end* block inside the task and not the task itself?[5] And what is missing from all those task implementations?[6]

CPU Operations

Encapsulated operations are also known as bus-functional models.

Operations encapsulated using *tasks* or *procedures* can be very complex. The examples shown earlier were very simple, and dealt with only a few signals. Real life interfaces are more complex. But they can be encapsulated just as easily. These operations may even

5. For the answer see "Output Arguments on Disabled Tasks" on page 150.
6. They all include timing control statements. They should have a semaphore to detect concurrent activation. See "Non-Reentrant Tasks" on page 151.

return values to be verified against expected values or modify the stimulus sequence.

Test vectors "hard code" a number of wait states.

Figure 5-18 shows the specification for the read cycle for an Intel 386SX processor bus. Being a synchronous interface, it could be verified using test vectors. However, you would have to assume a specific number of wait cycles to sample the read data at the right time.

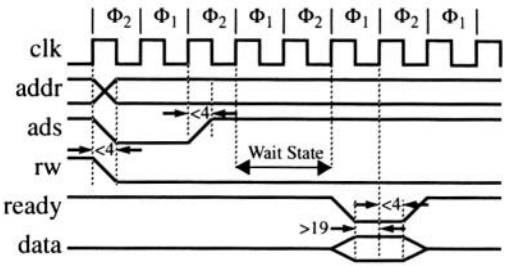


Figure 5-18. Specification for the read cycle of a 386sx processor

Bus models can adapt to a different number of wait states.

With behavioral models of the operation, you need not enforce a particular number of wait states and adapt to any valid bus timing. A model of the read operation can be found in Sample 5-46. The wait states are introduced by the fourth *wait* statement. How many failure modes are currently ignored by this model?[7]

Test vectors cannot perform read-modify-write operations.

In test vectors, the read value would have been specified as an expected output value. If that value had been different from the one specified, an error would have been detected. But what if you do not know the entire value that will be read? All you want is to modify the configuration of some slave device by reading its current configuration, modifying some bits, then writing the new configuration. This simple process is impossible to accomplish with test vectors blindly applied from a file to the inputs at every clock cycle. In behavioral testbenches, you can use the value returned during a read cycle, manipulate it, then use it in another operation. Sample 5-47 shows a portion of a testcase where the *read_cycle* procedure

7. Two: if `clk = '1' and phi = '2'` are never true and if *ready* is never asserted.

Sample 5-46. Model for the read cycle operation

```
procedure read_cycle (
          radd : in    std_logic_vector(0 to 23);
          rdat : out   std_logic_vector(0 to 31);
   signal clk  : in    std_logic;
   signal phi  : in    one_or_two;
   signal addr : out   std_logic_vector(0 to 23);
   signal ads  : out   std_logic;
   signal rw   : out   std_logic;
   signal ready: in    std_logic;
   signal data : inout std_logic_vector(0 to 31);
is
begin
   wait on clk until clk = '1' and phi = 2;
   addr <= radd after rnd_pkg.random * 4 ns;
   ads  <= '0'  after rnd_pkg.random * 4 ns;
   rw   <= '0'  after rnd_pkg.random * 4 ns;
   wait until clk = '1';
   wait until clk = '1';
   ads  <= '1' after rnd_pkg.random * 4 ns;
   wait on clk until clk = '1' and phi = 2 and
                       ready = '0';
   assert ready'stable(19 ns) and
          data'stable(19 ns);
   rdat := data;
   wait for 4 ns;
   assert ready = '1' and data = (others => 'Z');
end read_cycle;
```

shown in Sample 5-46 and its corresponding *write_cycle* procedure are used to perform a read-modify-write operation.

Sample 5-47. Performing a read-modify-write operation

```
test_procedure: process
   constant cfg_reg: std_logic_vector(0 to 23)
      := "000000000000000001100010110";
   variable tmp: std_logic_vector(31 downto 0);
begin
   ...
   i386sx_pkg.read_cycle(cfg_reg, tmp, ...);
   tmp(13 downto 9) := "01101";
   i386sx_pkg.write_cycle(cfg_reg, tmp, ...);
   ...
end process test_procedure;
```

Configurable Operations

Interfaces can have configurable elements.

An interface specification may contain configuration options. For example, the assertion level for a particular control signal may be configurable to either high or low. Each option has a small impact on the operation of the interface. Taken individually, you could create a different task or procedure for each configuration. The problem would be relegated to the testcase in deciding which flavor of the operation to invoke. You would also have to maintain several nearly-identical models.

Simple configurable elements become complex when grouped.

Taken together, the number of possible configurations explodes factorially.[8] It would be impractical to provide a different procedure or task for each possible configuration. It is much easier to include configurability in the model of the operation, and make the current configuration an additional parameter. An RS-232 interface, shown in Figure 5-19, is the perfect example of a highly configurable operation. Not only is the polarity of the parity bit configurable, but also its presence, as well as the number of data bits transmitted. And to top it all, because the interface is asynchronous, the duration of each pulse is also configurable. Assuming eight possible baud rates, five possible parities, seven or eight data bits, and one or two stop bits, there are 160 possible combinations of these four configurable parameters.

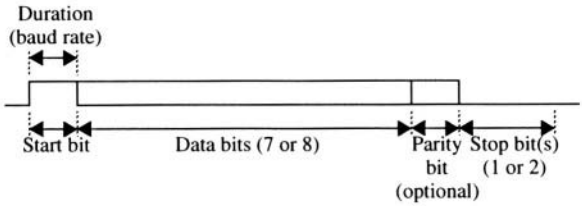


Figure 5-19. Specification for the RS-232 interface

Write a configurable operation model.

Instead of writing 160 flavors of the same operation, it is much easier to model the configurablity itself, as shown in Sample 5-48. The configuration parameter is assumed to be a record containing a field for each of the four parameters. Since Verilog does not directly support record types, refer to “Records” on page 105 for the implemen-

8. Exponential growth follows a K^n curve. Factorial growth follows a n! curve, where n! = 1 x 2 x 3 x 4 x ... x (n-2) x (n-1) x n.

tation details. What important safety measure is missing from the task in Sample 5-48?[9]

Sample 5-48. Model for a configurable operation

```
`define sec * 1000000000 // timescale dependent!
task rs232_tx;
   input [7:0]          data;
   input `rs232_cfg_typ cfg;

   time    duration;
   integer i;
begin
   duration = (1 `sec) / cfg`baud_rate;
   tx = 1'b1;
   #(duration);
   for (i = cfg`n_bits; i >= 0; i = i-1) begin
      tx = data[i];
      #(duration);
   end
   if (cfg`parity != `none) begin
      if (cfg`n_bits == 7) data[7] = 1'b0;
      case (cfg`parity)
      `odd  : tx = ~^data;
      `even : tx = ^data;
      `mark : tx = 1'b1;
      `space: tx = 1'b0;
      endcase
      #(duration);
   end
   tx = 1'b0;
   repeat (cfg`n_stops) #(duration);
end
endtask
```

COMPLEX RESPONSE

Output verification must be automated.

We have already established that visual inspection is not a viable option for verifying even a simple response. Complex responses are definitely not verifiable using visual inspection of waveforms. The process for verifying the output response must be automated.

9. The task contains timing control statements. It should contain a semaphore to detect concurrent activation. See “Non-Reentrant Tasks” on page 151.

Verifying response is usually not as simple as checking the outputs after each stimulus operation. In this section, I describe how complex monitors are implemented using bus-functional models. I show how to manage separate control threads to make a testbench independent of the latency or delay within a design. I explain how to design a series of output monitors to handle non-deterministic responses as well as bi-directional interfaces.

What is a Complex Response?

Latency and output protocols create complex responses.

I define a complex response as something that cannot be verified in the same process that generates the stimulus. A complex response situation could be created simply because the validity of the output cannot be verified at a single point in time. It could also be created by a long (and potentially variable) delay between the stimulus and the corresponding response. These types of responses cannot be verified as part of the stimulus generation because the input sequence would be interrupted while it waits for the corresponding output value to appear. Interrupting the input sequence would prevent stressing the design at the fastest possible input rate. Holding the input sequence may even prevent the output from appearing or violate the input protocol. A complex response must be verified autonomously from the stimulus generation.

A simple design can have a complex response.

A Universal Asynchronous Receiver Transmitter (UART) is a perfect example of a simple design with a complex response. And not only because the output operation is configurable. Figure 5-20 shows the block diagram of the transmit path. Because the RS-232 protocol is so much slower than today's processor interfaces, waiting for the output corresponding to the last CPU write cycle would introduce huge gaps in the input stimulus, as shown in Figure 5-21. The design would definitely not be verified under maximum input stress conditions.

Figure 5-20. Block diagram of a UART transmit path

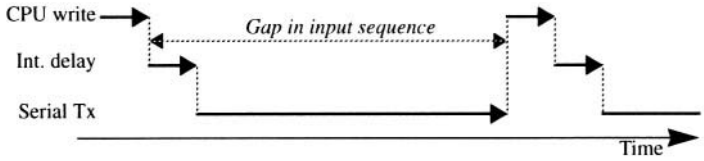


Figure 5-21. Stimulus and reponse checking in a single process

To stress the input of the design under maximum data rate condition, the testcase must decouple the input generation from the output verification. The stimulus generation part would issue write cycles as fast as it could, as long as the design can accept the data to be transmitted. It would stop generating write cycles only when the FIFO was full and the CPU interface signals that it can no longer accept data. The output would be verified in parallel, checking that the words that were sent to the design via the CPU interface were properly received via the serial interface. Figure 5-22 shows the timing of checking the output independently from generating the input. Gaps in the input sequence are created by the design's own inability to sustain the input rate, not by a limitation of the verification procedure.

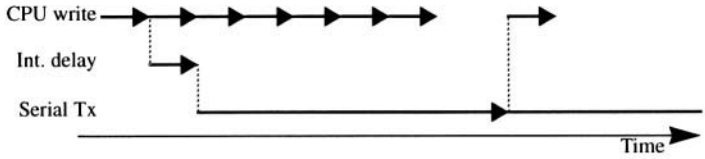


Figure 5-22. Stimulus and reponse checking in independent processes

Handling Unknown or Variable Latency

Test vectors cannot deal with variable latency.

When using test vectors, you have to assume a specific delay between the input and its corresponding output. The delay is expressed in terms of the number of clock cycles it takes for the input value to be processed into its output value. This delay is known as the *latency* of the design. Latency is usually a by-product of the architecture of the design and is a side-effect of the pipelining required to meet the register-to-register timing constraints. The specific latency of a design is normally known only toward the very end of the RTL design process. A specific latency is rarely a design requirement. If a specific latency is not a requirement, why enforce one in the verification?

Verify only the characteristics that define functional correctness.

In the UART design from Figure 5-20, latency is introduced by the CPU interface, the FIFO and the serial interface. Externally, it translates into the internal delay shown in Figure 5-21 and Figure 5-22. The latency of the UART design is functionally irrelevant. The functional correctness of the design is solely determined by the data being transmitted, unmodified, in the same order in which it was received by the CPU interface. Those are the only criteria the testbench should be verifying. Any other limitations imposed by the testbench would either limit the freedom of choice for the RTL designer in implementing the design, or turn into a maintenance problem for you, the testbench designer.

Stimulus and response are implemented using different execution threads.

Verification of the output independently from the stimulus generation requires that each be implemented in separate *execution threads.* Each must execute independently from the other, i.e., in separate parallel constructs (*processes* in VHDL, *always* or *initial* blocks and *fork/join* statements in Verilog). These execution threads need to be synchronized at appropriate points. Synchronization is required to notify the response checking thread that the stimulus generation thread is entering a different phase of the test sequence. It is also required when either thread has completed its duty for this portion of the test sequence and the other thread can move on to the next phase.

Sample 5-49. Using a named event in Verilog

```
event sync;
initial
begin: stimulus
    ...
    -> sync;
    ...
end

initial
begin: response
    ...
    @ (sync);
    ...
end
```

Synchronize threads using *fork/join* or named events in Verilog or signal activity in VHDL.

In Verilog, implicit synchronization occurs when using the *fork/join* statement. Explicit synchronization is implemented using the *named event,* as illustrated in Sample 5-49. In VHDL, synchronization is implemented using a toggling signal, as shown in Sample 5-50. The actual value of the boolean signal is irrelevant. The infor-

mation is in the timing of the value-change. Alternatively, the *'transaction* signal attribute can be used to synchronize a process with the assignment of a value to a signal, as shown in Sample 5-51. Pop quiz: why use the *'transaction* attribute and not simply wait for the event on the signal caused by the new value?[10]

Sample 5-50. Using a toggling boolean in VHDL

```
architecture test of bench is
   signal sync: boolean;
begin
   stimulus: process
   begin
      ...
      sync <= not sync;
      ...
   end process stimulus;
      ...
   response: process
   begin
      ...
      wait on sync;
      ...
   end process response;
end test;
```

Sample 5-51. Using the *'transaction* attribute in VHDL

```
architecture test of bench is
   signal expect: integer;
begin
   stimulus: process
   begin
      ...
      expect <= ...;
      ...
   end process stimulus;

   response: process
   begin
      ...
      wait on expect'transaction;
      ...
   end process response;
end test;
```

10. Because the assignment of the same value, twice in a row, will not cause an event on the signal. To synchronize to the assignment of any value on a signal, you must be sensitive to any assignment, even of values that do not cause an event. That's what the *'transaction* attribute identifies.

Figure 5-23 shows the execution threads for the verification of the UART transmit path. It also shows the synchronization points when the testcase switches from loading a new configuration in the UART to actively transmitting data, and vice-versa.

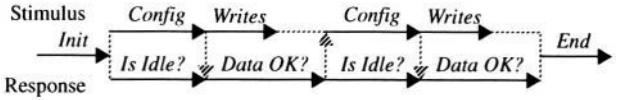


Figure 5-23. Execution threads for UART Tx testcase

The Verilog implementation of the execution threads shown in Figure 5-23 is detailed in Sample 5-52.

Sample 5-52. Implementing execution threads in Verilog

```
initial
begin
    ... // Init simulation

    fork: config_phase
       begin
           ... // Config
          disable config_phase
       end
       begin
           ... // Check output remains idle
       end
    join

    fork: data_phase
       begin
           ... // Write data to send via CPU i/f
       end
       begin
           ... // Check data sent serially
       end
    join

    ... // Terminate simulation
end
```

Controlling execution threads is simplified by using the *fork/join* statement.

The implementation in VHDL is a little more complex. Since VHDL lacks the *fork/join* statement, individual processes must be used. A process must be selected as the "master" process. The master process controls the synchronization of the various execution threads in the testcase. The master process can be a separate process whose sole function is to control the execution of the testcase. It

could also be one of the execution threads, usually one of the stimulus generation threads. An emulation of the *fork/join* statement, as shown in "Fork/Join Statement" on page 134, could be used. In Sample 5-53, a simple synchronization scheme using a different signal for each synchronization point is used.

Sample 5-53. Implementing execution threads in VHDL

```
architecture test of bench is
   signal sync1, syn2, sync3, done: boolean;
begin
   stimulus: process
   begin
       ... -- Init simulation
      sync1 <= not sync1;
      ... loop
          ... -- Config via CPU i/f
          sync2 <= not sync2;
          ... -- Write data to send via CPU i/f;
          -- Wait for data to be received
          wait on sync3;
      end loop;
      done <= true;
      wait;
   end process stimulus;

   response: process
   begin
      -- Wait until init is complete
      wait on sync1;
      loop
         -- Check output is idle while config
         wait until Tx /= '0' or
                    sync2'event or done;
         if done then
             -- Terminate simulation
             assert FALSE severity FAILURE;
         end if;
         assert Tx = '0' . . . ;
          ... -- Verify data sent serially
         sync3 <= not sync3;
      end loop;
   end process response;
end test;
```

Abstracting Output Operations

Output operations can be encapsulated.

Earlier in this chapter, we encapsulated input operations to abstract the stimulus generation from individual signals and waveforms to

generating sequences of operations. A similar abstraction can be used for verifying the output. The repetitiveness of output signals is taken care of and verified inside the subprograms. The output verification thread simply passes the expected output value to the monitor subprogram.

Arguments include expected value and configuration parameters.

The input operations takes as argument the specific value to use to generate the stimulus. Conversely, the output operations, encapsulated using *tasks* in Verilog, or *procedures* in VHDL, take as argumen the value expected to be produced by the design. If the format or protocol of the output operation is configurable, its implementation should be configurable as well.

A perfect example is the operation to verify the serial output in a UART transmit path, shown in Figure 5-20. It is as highly configurable as its input counterpart, detailed in Sample 5-48. The difference is that it compares the specified value with the one received, and compares the received parity against its expected value based on the received data and the configuration parameters. An implementation of the RS-232 receiver operation is detailed in Sample 5-54. It assumes that the configuration is specified using a user-

Sample 5-54. Implementation of the RS-232 serial receive operation

```
subtype byte is std_logic_vector(7 downto 0) ;
procedure recv(signal rx     : in std_logic;
                       expect: in byte;
                       config: in rs232_cfg_typ)
is
   variable period : time;
   variable actual: byte := (others => '0');
begin
   period := 1 sec / config.baud_rate;
   wait until rx = '1';   -- Wait for start bit
   wait for period / 2;   -- Sample mid-pulse
   for I in config.n_bits downto 0 loop
      wait for period;
      actual (I) := rx;    -- 7-8 data bits
   end loop;
   assert actual = expect; -- Compare
   -- Parity bit?
   if (config.parity /= no_parity) then
       wait for period;
       assert rx = parity(actual, config.parity);
   end
   wait for period;       -- Stop bit
   assert rx = '0';
end recv;
```

defined record type. A function to compute the parity of an array of bits, based on a configurable parity value, is also assumed to exist. This parity function could use the *xor_reduce* function available in Synopsys's *std_logic_misc* package.

Consider all possible failure modes.

The procedure shown in Sample 5-54 has some potential problems and limitations. What if the output being monitored is dead and the start bit is never received? This procedure will hang forever. It may be a good idea to provide a maximum delay to wait for the start bit via an additional argument, as shown in Sample 5-55, or to compute a sensible maximum delay based on the baud rate. Notice how a default argument value is used in the procedure definition to avoid forcing the user to specify a value when it is not relevant, as shown in Sample 5-56, or to avoid modifying existing code that was written before the additional argument was added.

Sample 5-55. Providing an optional timeout for the RS-232 serial receive operation

```
procedure recv(signal rx      : in std_logic;
                      expect  : in byte;
                      config  : in rs232_cfg_typ:
                      timeout: in time
                                   := TIME'high)
is
   ...
begin
   ..
   wait until rx = '1' for timeout;
   assert rx = '1';
   ...
end recv;
```

Sample 5-56. Using the RS-232 serial receive operation

```
process
begin
   ...
   recv(rx, "01010101", cfg_9600_8N1, 100 ms);
   recv(rx, "10101010", cfg_9600_8N1);
   ...
end process;
```

Do not arbitrarily constrain the operation.

The width of pulses is not verified in the implementation of the RS-232 receive operation in Sample 5-54. Should it? If you assume that the procedure is used in a controlled, 100 percent digital environment, then verifying the pulse width might make sense. This procedure could also be used in system-level verification, where the serial signal was digitized from a noisy analog transmission line as

illustrated in Figure 5-24. In that environment, the shape of the pulse, although unambiguously carrying valid data, most likely does not meet the rigid requirements of a clean waveform for a specific baud rate. Just as in real life, where modems fail to communicate properly if their baud rates are not compatible, an improper waveform shape is detected as invalid data being transmitted.

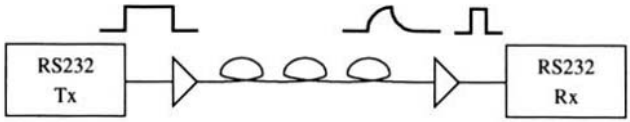


Figure 5-24. Modification to the serial signal in a real system

Generic Output Monitors

Verifying the value in the output monitor is too restrictive.

The output verification operation, as encapsulated in Sample 5-54, has a very limited application. It can be only used to verify that the output value matches a predefined expected value. Can you imag-

Sample 5-57. Generic RS-232 serial receive operation

```
subtype byte is std_logic_vector(7 downto 0);
procedure recv(signal rx     : in  std_logic;
                           actual: out byte;
                           config: in  rs232_cfg_typ)
is
   variable period: time;
   variable data  : byte;
begin
   period := 1 sec / config.baud_rate;
   wait until rx =  '1';   -- Wait for start bit
   wait for period / 2;    -- Sample mid-pulse
   data(7) := '0';         -- Handle 7 data bits
   for I in config.n_bits downto 0 loop
      wait for period;
      data(I) := rx;     --  7-8 data bits
   end loop;
   -- Parity bit?
   if (config.parity /= no_parity) then
      wait for period;
      assert rx = parity(data, config.parity);
   end
   wait for period;        -- Stop bit
   assert rx = '0';
   actual := data;
end recv;
```

ine other possible uses? What if the output value can be any value within a predetermined set or range? What if the output value is to be ignored until a specific sequence of output values is seen? What if the output value, once verified, needs to be fed back to the stimulus generation? The usage possibilities are endless. It is not possible, a priori, to determine all of them nor to provide a single interface that satisfies all of their needs.

Separate monitoring from value verification.

The most flexible implementation for an output operation monitor is to simply return to the caller whatever output value was just received. It will be up to a "higher authority" to determine if this value is correct or not. The RS-232 receiver was modified in Sample 5-57 to return the byte received without verifying its correctness. The parity, being independent of the correctness of the value and fully contained within the RS-232 procotol, can still be verified in the procedure.

Monitoring Multiple Possible Operations

The next operation on an output interface may not be predictable.

You may be in a situation where more than one type of operation can happen on an output interface. Each would be valid and you cannot predict which specific operation will come next. An example would be a processor that executes instructions out of order. You cannot predict (without detailed knowledge of the processor architecture) whether a read or a write cycle will appear next on the data memory interface. The functional validity is determined by the proper access sequence to related data locations.

Sample 5-58. Processor test program

```
load A,  R0
load B,  R1
add  R0, R1, R2
sto  R2, X
load C,  R3
add  R0, R3, R4
sto  R4, Y
```

Verify the sequence of related operations.

For example, consider the testcase composed of the instructions in Sample 5-58. It has many possible execution orders. From the perspective of the data memory, the execution is valid if the conditions listed below are true.

1. Location A is read before location X and Y are written.
2. Location B is read before location X is written.

3. Location C is read before location Y is written.
4. Location X must be written with the value A+B.
5. Location Y must be written with the value A+C.

These are the sufficient and necessary conditions for a proper execution of the test program. Verifying for a particular order of the individual cycle overconstrains the testcase.

Write an operation "dispatcher" task or procedure.

How do you write an encapsulated output monitor when you do not know what kind of operation comes next? You must first write a monitor that identifies the next cycle after it has started. It verifies the preamble to all operations on the output interface until it becomes unique to a specific operation. It then returns any information collected so far and identifies, to the testbench, which cycle is currently underway. It is up to the testbench to then call the appropriate *task* or *procedure* to complete the verification of the operation.

Sample 5-59 shows the skeleton of a monitor task that identifies whether the next operation for a CPU is a read or a write cycle. Since the address has already been sampled by the time the decision of the type of cycle was made, it is returned along with the current cycle type. Sample 5-60 shows how this operation identification task is used by the testbench to determine the next course of action.

Sample 5-59. Monitoring many possible output operations

```
parameter READ_CYCLE  = 0,
          WRITE_CYCLE = 1;
time last_addr;
task next_cycle_is;
   output         cycle_kind;
   output [23:0] address;
begin
   @ (negedge ale);
   address = addr;
   cycle_kind =
       (rw == 1'b1) ? READ_CYCLE : WRITE_CYCLE;
   #(Tahold);
   if ($time - last_addr < Tahold + Tasetup)
      $write("Setup/Hold time viol, on addr\n");
end
endtask

always @ (addr) last_addr = $time;
```

Sample 5-60. Handling many possible output operations

```
initial
begin: test_procedure
   reg         cycle_kind;
   reg  [23:0]  addr;

   next_cycle_is(cycle_kind, addr);
   case  (cycle_kind)
   READ_CYCLE:    read_cycle(addr);
   WRITE_CYCLE: write_cycle(addr);
   endcase
   ...
end
```

In this case, we assume the existence of two tasks, one for each possible operation, which completes the monitoring of the remainder of the cycle currently under way.

Monitoring Bi-Directional Interfaces

Output interfaces may need to reply with "input" data.

We have already seen that input operations sometimes have to monitor some signals from the design under verification. The same is true for output monitor. Sometimes, they have to provide data back as an answer to an "output" operation. This blurs the line between stimulus and response. Isn't a stimulus generation subprogram that verifies the control or feedback signals from the design also doing response checking? Isn't a monitor subprogram that replies with new data back to the design also doing stimulus generation?

Generation and monitoring pertains to the ability to initiate an operation.

The terms *generator* and *monitor* become meaningless if they are attached to the direction of the signals being generated or monitored. They regain their meaning if you attach them to the initiation of operations. If a procedure or a task *initiates* the operation, it is a *stimulus generator.* If the procedure or task sits there and waits for an operation *to be initiated by the design,* then it is an *output monitor.* The latter also includes ancilliary tasks to complete a cycle currently underway, as discussed in "Monitoring Multiple Possible Operations" on page 203.

Bridges have bi-directional output interfaces.

The downstream master interface on a bus bridge is the perfect example of a bi-directional "output" interface. The bridge is the design under verification. An example, illustrated in Figure 5-25, is a bridge between a proprietary on-chip bus and a PCI interface. The cycles are initiated on the on-chip bus (upstream). If the address falls within the bridge's address space, it translates the cycle onto

the PCI bus (downstream). This bridge allows master devices on the on-chip bus to transparently access slave devices on the PCI bus.

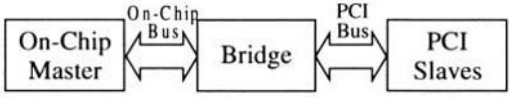


Figure 5-25. Bridge between an on-chip bus and a PCI bus

Using a memory to test a CPU interface can mask certain classes of problems.

To verify this bridge, you would need an on-chip bus cycle generator and a PCI bus cycle monitor, as illustrated in Figure 5-26. Many would be tempted to use a model of a memory (which, for PCI, are readily available from model suppliers), instead of a PCI monitor. The verification would be accomplished by writing a pattern in the memory then reading it back. Using a memory would not catch several types of problems masked by the readback operations.

For example, what if the bridge designer misreads the PCI specification document and implements the address bus using little endian instead of big endian? During the write cycle, address 0xDEADBEEF on the on-chip bus is translated to the physical address 0xEFBEADDE on the PCI bus, writing the data in the wrong memory location. The read cycle, used to verify that the write cycle is correct, also translates address 0XDEADBEEF to 0xEFBEADDE and reads the data from the same, but invalid location. The testbench does not have the necessary visibility to detect the error.

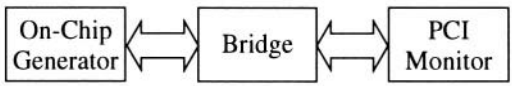


Figure 5-26. Verification structure for the bridge

Use a monitor that detects the PCI cycles and notifies the testbench.

Using a generic PCI monitor to verify the output detects errors that would be masked by using a write-readback process. The PCI monitor task or procedure would watch the bus until it determined the type of cycle being initiated by the bridge. To ease implementation, this task or procedure usually continues to monitor the bus while the cycles remain identical (i.e. for the entire address phase). Assuming that the bridge's implementation is limited to generating

Assuming that the bridge's implementation is limited to generating PCI memory read and write cycles, the monitor task or procedure would then return, identifying the cycle as a memory read or write and the address being read or written. The skeleton for a PCI bus monitor is shown in Sample 5-61.

Sample 5-61. Monitoring many possible PCI cycles

```
parameter MEM_RD = 0,
          MEM_WR = 1;
task next_pci_cycle_is;
   output        cycle_kind;
   ouptut [31:0] address;
begin
   // Wait for the start of the cycle
   @ (posedge pci_clk);
   while (frame_n !== 1'b0) @ (posedge pci_clk);
   // Sample command and address
   case (cbe_n)
   4'b0110: cycle_kind = MEM_RD;
   4'b0111: cycle_kind = MEM_WR;
   default: $write("Unexpected cycle type!\n");
   endcase
   address = ad;
end
endtask
```

You must be able to verify different possible answers by the bus monitor.

The really interesting part comes next. In PCI, read and write cycles can handle arbitrary length bursts of multiple data values in a single cycle. A read cycle can read any number of consecutive bytes and a write cycle can write any number of consecutive bytes. The PCI master is under control of the length of the burst, but the number of bytes involved in a cycle is not specified in the preamble. Data must be read or written by the slave for as long as the master keeps reading or writing them.

In a VHDL procedure, implementing a monitor for the write cycle, you could use an access value to an array of bytes to return all of the data values that were written during a burst. The instance of the array object would be dynamically allocated with the proper constraints according to the number of bytes read. An example is shown in Sample 5-62. But what about read cycles where the testbench cannot know, a priori, how many bytes will be read? And what about Verilog which does not support arrays of bytes on interfaces, let alone unconstrained arrays?

Sample 5-62. Monitoring burst write cycles in VHDL

```
subtype byte is std_logic_vector(7 downto 0);
type byte_array_typ is array(natural range <>)
   of byte;
type burst_data_typ is access byte_array_typ;

procedure next_pci_cycle_is(...);

procedure pci_mem_write_cycle(
   -- PCI bus interface as signal class formals
   signal  pci_clk: in  std_locic;
   ...
   -- Pointer to data values written during cycle
   variable data_wr: out burst_data_typ);
```

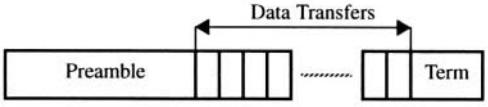
A monitor can be composed of several *tasks* or *procedures* that must be appropriately called by the testbench.

The PCI bus monitor has been implemented by slicing the PCI cycles into at least two procedures: one to handle the generic preamble and detect the type of cycle under way, the other intended to handle the remainder of each cycle.

The solution to our dilemma is to slice the implementation of the PCI bus monitor even further: use a procedure to handle each data transfer and one to handle the cycle termination.

The data transfer procedure or task would have an input or output argument for the data read or written, and an output argument to indicate whether to continue with more data transfers or terminate the cycle. Figure 5-27 illustrates how each procedure is sequenced, under the control of the testbench, to form a complete PCI cycle. A similar slicing strategy would be used in creating a PCI bus generator. The only exception being that the generator is now under the control of the initiation of the cycle and its duration.

Figure 5-27. Slicing the PCI cycle into procedures



Provide the controls at the proper level of granularity.

Slicing the implementation of the PCI cycle at the data transfer level offers an additional opportunity. In PCI, both master and slave can throttle the transfer rate by asserting the *irdy_n* and *trdy_n* signals, respectively, when they are ready to complete a transfer.

A procedure or a task implementing a single data transfer can have an additional parameter specifying the number of clock cycles to wait before asserting the *trdy_n* signal. Another could be used to specify whether to force one of the target termination exception. It can also report that a master-initiated exception occured during this data transfer, such as a timeout, a system error, or a parity error.

The testbench would then be free to "generate" any number of possible answers to a PCI cycle. With the tasks outlined in Sample 5-61 and Sample 5-63, the possibilities become endless! One of these possibilities is shown in Sample 5-64.

Sample 5-63. PCI data transfer and termination tasks

```
// Target terminations
parameter NORMAL     = 0,
          RETRY      = 1,
          DISCONNECT = 2,
          ABORT      = 3;
// Output status
parameter TERMINATE        = 0,
          CONTINUE         = 1,
          INITIATOR_ABORT  = 2,
          PARITY_ERROR     = 3,
          SYS_ERROR        = 4;
task pci_data_rd_xfer;
   input [31:0] rd_data;
   input [ 7:0] delay;
   input [ 1:0] termination;
   output [  2:0] status;
...
endtask

task pci_data_wr_xfer;
   output [31:0] wr_data;
   input  [ 7:0] delay;
   input  [ 1:0] termination;
   output [ 2:0] status;
...
endtask

task pci_end_cycle;
   output [2:0] status;
...
endtask
```

Sample 5-64. Monitoring a complete PCI cycle

```
initial
begin: test_procedure
   fork
      begin: on_chip_side
         // Generate a long read cycle on the
         // On-Chip bus side
         ...
      end
      begin: pci_side
         reg        kind;
         reg [31:0] addr;
         integer    delay;
         integer    ok;

         // Expect a read cycle on the PCI side
         // at the proper address
         next_pci_cycle_is(kind, addr);
         if (kind != MEM_RD) ...
         if (addr !=...) ...

         // Send back 5 random data words
         // with increasing delays in readiness
         // then force a target abort on the 6th.
         delay = 0;
         repeat (5) begin
            pci_data_rd_xfer($random, delay,
                             NORMAL, ok);
            if (ok !== CONTINUE) ...
            delay = delay + 1;
         end
         pci_data_rd_xfer($random, 0, ABORT, ok);
      end
   join
end
```

Using a monitor simplifies the testcase.

Using the generic PCI bus monitor also shortens the testcase compared to using a memory. With the monitor, you have direct access to all of the bus values. It is not necessary to write into the memory for the entire range of address and data values, creating interesting test patterns that highlight potential errors. With a monitor, only a few addresses and data values are sufficient to verify that the bridge properly translates them. It is also extremely difficult to control the answers provided by the memory to test how the bridge reacts to bus exceptions. These exception conditions become easy to setup with a generic monitor designed to create them.

PREDICTING THE OUTPUT

The unstated assumption in implementing self-checking testbenches is that you have detailed knowledge of the output to be expected. Knowing exactly which output to expect and how it can be verified to determine functional correctness is the most crucial step in verification. In some cases, such as RAMs or ROMs, the response is easy to determine. In others, such as a video compressor or a speech synthesizer, the response is much more difficult to define. This section examines various families of designs and show how the expected response could be determined and communicated to the output monitors.

Data Formatters

The expected output equals the input.

There is a class of designs where the input information is not transformed, but simply reformated. Examples include UARTs, bridges, and FIFOs. They have the simplest output prediction process. Since the information is not modified, predicting the output is a simple matter of knowing the sequence of input values.

Forwarding one value at a time under-constrains the design.

Passing data values, one at a time, from the stimulus generator to the response monitor, as illustrated in Figure 5-28, is usually not appropriate. This limits the data rate to one every input and output cycle and may not stress the design under worse conditions. Pipelined designs cannot be verified using this stategy: input must be continuously supplied while their corresponding response has not yet appeared on the output.

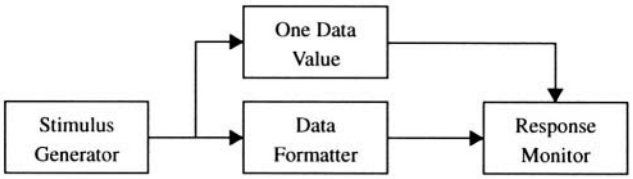


Figure 5-28. Forwarding one value at a time

A short data sequence can be implemented in a global array.

If the input sequence is short and predetermined, using a global data sequence table is the simplest approach. Both the input generator and output monitor use the global table. The input generator applies each value in sequence. The output monitor compares each output value against the sequence of values in the global table. Figure 5-29

illustrates the flow of information while Sample 5-65 shows an implementation in VHDL.

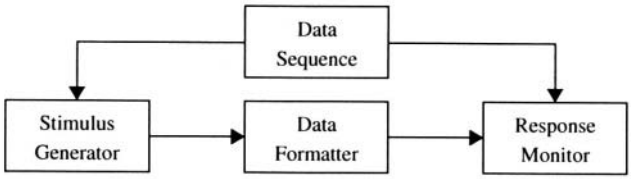


Figure 5-29. Using a global table to predict output.

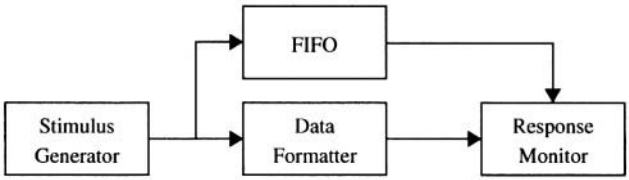
Sample 5-65. Implementation of a global data sequence table

```
architecture test of bench is
   type std_lv_ary_typ is array(natural range <>)
      of std_logic_vector(7 downto 0);
   constant walking_ones: std_lv_ary_typ(1 to 8)
      := ("10000000",
          "01000000",
          "00100000",
          "00010000",
          "00001000",
          "00000100",
          "00000010",
          "00000001");
begin
   DUV: ...

   stimulus: process
   begin
      for I in walking_ones'range loop
         apply(walking_ones(I), . . . );
      end loop;
      wait;
   end process stimulus;

   response: process
   begin
      for I in walking_ones'range loop
         expect(walking_ones(I), ...);
      end loop;
      assert false severity failure;
   end process response;
end test;
```

Long data sequence can use a FIFO between the generator and monitor.

Often the input sequence is long or computed on-the-fly. It is not practical for hardcoding in a global constant. A FIFO can be used to forward expected values from the stimulus generator to the output monitor. The input generator puts each value in sequence in one end of the FIFO. The output monitor compares each output value against the sequence of values dequeued from the other end of the FIFO. This strategy is a simple extension of the concept of forward-

Figure 5-30. Forwarding data values via a FIFO



Sample 5-66. Implementation using a FIFO to forward data values

```
task put_fifo;
   ...
endtask

function [7:0] get_fifo;
   ...
endfunction

initial
begin: stimulus
   reg [7:0] data;

   repeat (...) begin
      data = ...;
      put_fifo(data);
      apply(data);
   end
end

initial
begin: response
   reg [7:0] data;

   repeat (...) begin
      data = get_fifo(0);
      expect(data);
   end
   $finish;
end
```

ing a single value at a time. It is illustrated in Figure 5-30. Notice how the architecture of the testbench is identical to the one illustrated in Figure 5-28. The code in Sample 5-66 shows the implementation structure in Verilog of a testbench using a FIFO. The implementation of the FIFO itself is left as an exercise to the reader.

The stimulus and response processes can read the same data file.

Sometimes, the data sequence is externally generated and supplied to the testbench using a data file. A file can be read, concurrently, by more than one process. Thus, the stimulus generator and response monitor can both read the file, using it in a fashion similar to a global array. The code in Sample 5-67 illustrates how this strategy could be implemented in VHDL. The filename is assumed to be

Sample 5-67. Implementation using an external data file

```
entity bench is
   generic (datafile: string);
end bench;

architecture test of bench is
begin
   DUV:  ...

   stimulus: process
      file infile  :  text is in datafile;
      variable L   : line;
      variable dat: std_logic_vector(7 downto 0);
   begin
      while not endfile(infile) loop
         readline(infile, L);
         read(L, dat);
         apply(dat, ...);
      end loop;
      wait;
   end process stimulus;

   response: process
      file infile : text is in datafile;
      variable L  : line;
      variable dat: std_logic_vector(7 downto 0);
   begin
      while not endfile(infile) loop
         readline(infile, L);
         read(L, dat);
         expect(dat, ...);
      end loop;
      assert false severity failure;
   end process response;
end test;
```

passed to the testbench via the command line using a generic of type *string*.

Packet Processors

Packets have untouched data fields.

This family of designs uses some of the input information for processing, sometimes transforming it. But it leaves portions of the input untouched and forwards it, intact, all the way through the design to an output. Examples abound in the datacom industry. They include Ethernet hubs, IP routers, ATM switches, and SONET framers.

Use the untouched fields to encode the expected tranformation.

The portion of the data input that passes, untouched, through the design under verification can be put to good use. It is often called *payload* and the term *packet* or *frame* is often used to describe the unit of data processed by the design. You must first determine, through a proper testcase, that the payload information is indeed not modified by the design. Subsequently, it can be used to describe the expected output for this packet. For each packet received, the output monitor uses the information in the payload to determine if it was appropriately processed.

This simplifies the testbench control structure.

Figure 5-31 shows the structure of a testbench for a four-input and four-output packet router. Notice how the output monitors are completely autonomous. This type of design usually lends itself to the simplest testbench control structures, assuming that the output monitors are sufficiently intelligent. The control of this type of testbench is simple because all the processing (stimulus and generation of expected response) is performed in a single location: the stimulus generator. Some minor orchestration between the generators may be required in some testcases when it is necessary to synchronize traffic patterns to create interesting scenarios.

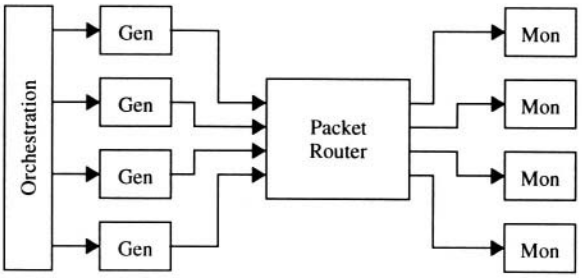


Figure 5-31. Testbench structure for a 4x4 packet router

Include all necessary information in the payload to determine functional correctness.

The payload must contain all necessary information to determine if a particular packet came out of the appropriate output, and with the appropriate transformation of its control information.

For example, assume the success criteria is that the packets for a given input stream be received in the proper order by the proper output port. The payload should contain a unique stream identifier, a sequence number, and an output port identifier, as shown in Figure 5-32.

The output monitor needs to verify that the output identifier matches its own identifer. It also needs to verify that the sequence number is equal to the previously-received sequence number in that stream plus one, as outlined in Sample 5-68. The Verilog records are assumed to be implemented using the technique shown in “Records” on page 105.

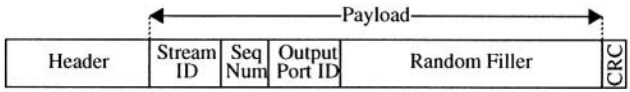


Figure 5-32. Example packet payload structure

Sample 5-68. Implementation using payload information to determine functional correctness

```
always
begin: monitor
   reg 'packet_typ pkt;

   receive_packet(pkt);
   // Packet is for this port?
   if (pkt'out_port_id !== my_id) ... ;
   // Packet in correct sequence?
   if (last_seq[pkt'strm_id] + 1
        != pkt'seq_num) ...;
   // Reset sequence number
   last_seq[pkt'strm_id] = pkt'seq_num;
end
```

Complex Transformations

The last family of designs processes and transforms the input data completely and thoroughly. The expected output can be only determined by reproducing the transformation using alternative means. This includes reversing the process where you determine which input sequence to provide in order to produce a desired output.

Use a truly alternative computation method.

When reproducing the transformation, to determine which output value to expect, as illustrated in Figure 5-33, you must use a different implementation of the transformation algorithm. For example, you can use a reference C model. For a DSP implementation, you could use floating-point expressions and the predefined *real* data types to duplicate the processing that is performed using fixed-point operators and data representation in the design.

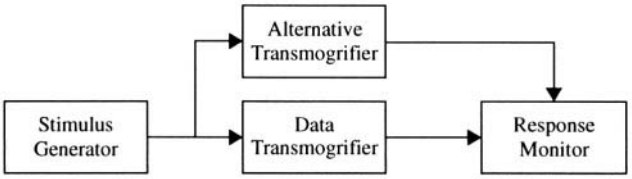


Figure 5-33. Reproducing the transformation to predict output

Use a different programming model for the output monitor.

If you are providing an input sequence to produce a specific output pattern, use a different programming model for the output monitor. The programming model for the design was chosen to ease implementation - or even to make it possible. Having almost no constraints in behavioral HDL models, you can choose a programming model that is more natural, to express the expected output. Using a different programming model also forces your mind to work in a different way when specifying the input and output, creating an alternative verification path.

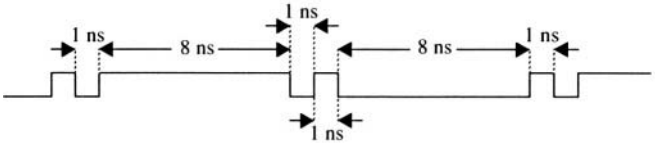


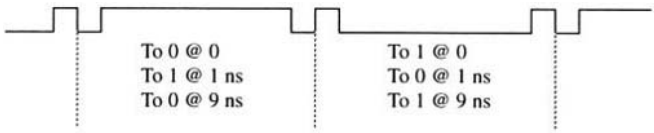
Figure 5-34. Target waveform to generate

Example: a waveform generator.

For example, you could be verifying a design that generates arbitrary digital waveforms. The input could specify, for each clock cycle, the position of up to three rising or falling edges within the clock period. Each transition is specified using two parameters. A *level* bit indicates the final logic level after the transition and a 10-bit *offset* value indicates the position of the transitions within the 10 ns clock period, with a resolution of 9.7 ps (or 10 ns / 1024). Assuming that the waveform in Figure 5-34 represents an interesting testcase, Figure 5-35 shows how it is sliced to create the input

sequence and Sample 5-69 shows how the stimulus could be generated.

Figure 5-35. Slices for input specification



Sample 5-69. Generating the input for the waveform generator

```
initial
begin: stimulus
   repeat (10) begin
      apply(1'b0, $realtobits(0.0),
            1'b1, $realtobits(1.0) ,
            1'b0, $realtobits(9.0)) ;
      apply(1'b1, $realtobits(0.0),
            1'b0, $realtobits(1.0),
            1'b1, $realtobits(9.0));
   end
end
```

Choose a different but reliable way of representing the output.

How should the output be represented? If we use a slicing method similar to the input's, it would not provide for an alternative programming model. Furthermore, the implementation could miss transitions or events between slices. The output waveform has no relationship with the clock. Trying to specify the expected output using clock-based slices would simply over-constrain the test. The validity of the output waveform is entirely contained in the relative position of the edges. So why not specify the expected output using

Sample 5-70. Monitoring the generated waveform

```
monitor: process
begin
   wait until wave = '1';
   for I in 1 to 10 loop
      wait on wave;
      assert wave'delayed'last_event = 8 ns;
      wait on wave;
      assert wave'delayed'last_event = 1 ns;
      wait on wave;
      assert wave'delayed'last_event = 1 ns;
   end loop;
   assert false severity failure;
end process monitor;
```

an edge-to-edge specification? Assuming that the output is initialized to a level of '0', an implementation of the output monitor is shown in Sample 5-70.

The *'delayed* attribute must be used to look before the wait statement.

The *'delayed* signal attribute must be used, otherwise *'last_event* always returns 0 since the signal *wave* just had an event to resume the execution of the previous *wait* statement. The *'delayed* attribute delays the *wave* signal by one delta cycle. The delayed *wave* signals looks to the *'last_event* attribute as if it were *before* the execution of the *wait* statement. Notice how this monitor simply waits for the first rising edge of the output monitor to anchor its edge-to-edge relationships. This makes the monitor completely independent of the latency and intrinsic delays in the design.

Do not enforce unnecessary precision.

There is one problem with the verification of the delay between edges in Sample 5-70. Each delay is compared to a precise value. However, the design has a resolution of 9.7 ps. Each delay is valid if it falls in the range of the ideal delay value plus or minus the resolution, as shown in Sample 5-71.

Sample 5-71. Handling uncertainty in the generated waveform

```
monitor: process
   function near(val, ref: in time)
      return boolean is
      constant resolution: time := 9700 fs;
   begin
      return ref - resolution <= val and
             val <= ref + resolution;
   end near;
begin
   wait until wave = '1';
   for I in 1 to 10 loop
      wait on wave;
      assert near(wave'delayed'last_event, 8 ns);
      wait on wave;
      assert near(wave'delayed'last_event, 1 ns);
      wait on wave;
      assert near(wave'delayed'last_event, 1 ns) ;
   end loop;
   assert false severity failure;
end process monitor;
```

SUMMARY

In this chapter, I have described how to use bus-functional models to generate stimulus and monitor response. The bus-functional

models were used to translate between high-level data representations and physical implementation levels. They also abstracted the interface operations, removing the testcases from the detailed implementation of each physical interface. Some of these bus-functional models can be very complex, depending on feed-back from the device under verification to operate properly or having to supply handshake information back to the device.

This chapter, after highlighting the problems with visual inpection, also described how to make each individual testbench completely self-checking. The expected response must be embedded in the testbench at the same time as the stimulus. Various strategies for determining the expected response and communicating it to the output monitors have been presented.

CHAPTER 6 ARCHITECTING TESTBENCHES

A testbench need not be a monolithic block. Although Figure 1-1 shows the testbench as a big thing that surrounds the design under verification, it need not be implemented that way. The design is also shown in a single block and it is surely not implemented as a single unit. Why should the testbench by any different?

The previous chapter was about low-level testbench components.

In Chapter 5, we focused on the generation and monitoring of the low-level signals going into and coming out of the device under verification. I showed how to abstract them into operations using bus-functional models. Each were implemented using a *procedure* or a *task*. The emphasis was on the stimulus and response of interfaces and the need for managing separate execution threads. If you prefer a bottom-up approach to writing testbenches, I suggest you start with the previous chapter.

This chapter focuses on the structure of the testbench.

This chapter concentrates on implementing the many testbenches that were identified in your verification plan. I show how to best structure the stimulus generators and response monitors to minimize maintenance, facilitate implementing a large number of testbenches, and promote the reusability of verification components.

REUSABLE VERIFICATION COMPONENTS

This section describes how to plan the architecture of testbenches. The goal is to maximize the amount of verification code reused across testbenches to minimize the development effort. The test-

Sample 6-1. Implementing the muxed flip-flop testbench in Verilog

```
module testbench;

reg rst, d0, d1, sel, clk;
wire q, qb;

muxed_ff duv(d0, d1, sel, q, qb, clk, rst);

parameter cycle  = 100,
          Tsetup =  15,
          Thold  =   5;

always
begin
   #(cycle/2) clk = 1'b0;
   #(cycle/2) clk = 1'bl;
end

task sync_reset;
   ...
endtask

task load_d0;
   input data;
begin
   rst <= 1'b0;
   d0  <= data;
   dl  <= ~data;
   sel <= 1'b0;
   @  (posedge clk);
   #(Thold);
   if (q !== data || qb !== ~data) ...
   {rst, d0, d1, sel} <= 4'bxxxx;
   #(cycle - Thold - Tsetup);
end
endtask

task load_d1;
   ...
endtask

initial
begin: test_sequence
   sync_reset;
   load_d0(1'b1);
   ...
   $finish;
end
endmodule
```

Sample 6-2. Implementing the muxed flip-flop testbench in VHDL

```
architecture test of bench is
   signal rst, d0, d1, sel, q, qb: std_logic;
   signal clk: std_logic := '0';
   component muxed_ff
      ...
   end component;
   constant cycle : time := 100 ns;
   constant Tsetup: time :=  15 ns;
   constant Thold : time :=   5 ns;
begin
   duv: muxed_ff port map(d0, d1, sel, q, qb,
                          clk, rst);

   clock_generator: clk <= not clk after cycle/2;

   test_procedure: process
      procedure sync_reset is
         ...
      end sync_reset;

      procedure load_d0(data: in std_logic) is
      begin
         rst <= '0' ;
         d0  <= data;
         d1  <= not data;
         sel <= '0' ;
         wait until clk = '1';
         wait for Thold;
         assert q == data and qb = not data;
         rst <= 'X';
         d0  <= 'X';
         d1  <=  'X';
         sel <= 'X';
         wait for cycle - Thold - Tsetup;
      end load_d0;

      procedure load_d1(data: in std_logic) is
         ...
      end load_d1;
   begin
      sync_reset;
      load_d0('1');
      ...
      assert FALSE severity FAILURE;
   end process test_procedure;
end test;
```

benches are divided into two major components: the reusable test harness, and the testcase-specific code.

Bus-functional models were assumed to be in the same process or module as the testbench.

In the previous chapter, stimulus generation and response checking were performed by abstracting operations using *procedures* or *tasks.* It was implied that these subprograms were implemented in the same *process* or *module* as the test sequence using them. Sample 6-1 shows where the task *load_d0,* first introduced in Sample 5-22, would have to be implemented in a Verilog testbench. Sample 6-2 shows the equivalent VHDL implementation.

Global access to signals declared at the module or architecture level was allowed.

With the suprograms located in the testbench module or process, they can be used by the *initial* and *always* blocks or processes implementing the testcase. The subprograms are simple to implement because the signals going to the device under verification can be driven directly through assignments to globally visible signals. Similarly, the outputs coming out of the device can be directly sampled as they too are globally visible.

The bus-functional models can be used by many testbenches.

All of the testbenches have to interface, through an instantiation, to the same design under verification. It is safe to assume that they all require the use of the same bus-functional models used to generate stimulus and to monitor response. These bus-functional models could be reused by all testbenches implemented for this design. If the interfaces being exercised or monitored by these bus-functional models have common interfaces found on other designs, they could even be reused by all testbenches for these other designs.

Use a low-level layer of reusable bus models.

Instead of a monolithic block, the testbenches should be structured with a low-level layer of reusable bus-functional models. This low-level layer is common to all testbenches for the design under verification and called the *test harness.* Each *testcase* would be implemented on top of these bus-functional models, as illustrated in Figure 6-1. The *testcase* and the *harness* together form a *testbench.*

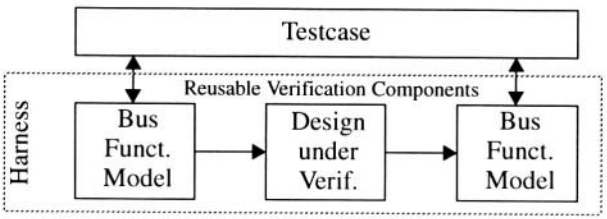


Figure 6-1. Structure of a testbench with reusable bus-functional models

Insert reusable mid-level utility routines as required.

Many testbenches share some common functionality or need for interaction with the device under verification. Once the low-level features are verified, the repetitive nature of communicating with the device under verification can also be abstracted into higher-level utility routines. For example, low-level read and write operations to send and receive individual bytes can be encapsulated by utility routines to send and receive fixed-length packets. These, in turn, can be encapsulated in a higher-level utility routine to exchange variable-length messages with guaranteed error-free delivery.

The testcase can operate at the required level of abstraction.

A testcase verifying the low-level read and write operations would interface directly with the low-level bus-functional model, as shown in Figure 6-1. But once these basic operations are demonstrated to function properly, testbenches dealing with higher-level functions can use the higher-level utility routines, as shown in Figure 6-2.

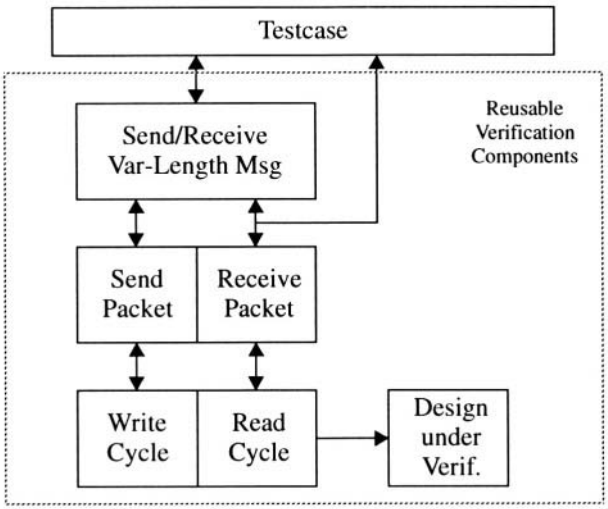


Figure 6-2. Structure of a testbench with reusable utility routines

Procedural Interface

Define a procedural interface to the bus-functional model and utility routines.

For these verification components to be reusable by many testbenches, you must define a procedural interface independent of their detailed implementation. A procedural interface simply means that all the functionality of these components is accessed through

procedures or tasks, never through global variables or signals. It is similar to providing tasks or procedures to encapsulate operations. This gives flexibility in implementing or modifying the bus-functional models and utility routines without affecting the testcases that use them.

Provide flexibility through thin layers.

The verification components need to be flexible enough to provide the required functionality for all testbenches that use them. It is better to provide this flexibility by layering utility routines on top of general purpose lower-level routines and bus-functional models. This approach creates layers of procedural interfaces. The low-level layer provides detailed control whereas the higher-level provides greater abstraction. Do not attempt to implement all functionality in a single level. It would unduly complicate the implementation of the bus-functional models and increase the risk of introducing a functional failure.

Preserve the procedural interfaces.

By stimulating and monitoring a design through procedural interfaces, it removes the testcase from knowing the low-level details of the physical interfaces on the design. If the procedural interface is well-designed and can support different physical implementations, the physical interface of a design can be modified without having to modify any testbenches.

For example, a processor interface could be changed from a VME bus to a X86 bus. All that needs to be modified is the implementation of the CPU bus-functional model. If the procedural interface to the CPU bus-functional model is not modified, none of the testbenches need to be modified.

Another example would be changing a data transmission protocol from parallel to serial. As long as the testcases can still send bytes, they need not be aware of the change. Once you have defined a procedural interface, document it and hesitate to change it.

Development Process

Introduce flexibility as required.

When developing the low-level bus-functional models and the utility routines, do not attempt to write the ultimate verification component that includes every possible configuration option and operating mode. Use the verification plan to determine the functionality that is ultimately required. Architect the implementation of the verification component to provide this functionality, but imple-

ment incrementally. Start with the basic functions that are required by the basic testbenches. As testbenches progress toward exercising more complex functions, develop the required supporting functions by adding configurability to the bus-functional models or creating utility routines. As the verification infrastructure grows, the procedural interfaces are maintained to avoid breaking testbenches already completed.

Incremental development maximizes the verification efficiency.

This incremental approach minimizes your development effort: you won't develop functionality that turns out not to be needed. You also minimize your debugging effort, as you are building on functionality that has already been verified and debugged with actual testbenches. This approach also allows the development of the verification infrastructure to parallel the development of the testbenches, removing it from the critical path.

VERILOG IMPLEMENTATION

This section evolves an implementation of the test harness and testbench architecture. Starting with a monolithic testbench, the implementation is refined into layers of bus-functional models, utility packages, and testcases, with well-defined procedural interfaces. The goal is to obtain a flexible implemention strategy promoting the reusability of verification components. This strategy can be used in most Verilog-based verification projects.

Creating another testcase simply requires changing the control block.

In Sample 6-1, the entire testbench is implemented in a single level of hierarchy. If you were to write another testcase for the muxed flip-flop design using the same bus-functional models, you would have to replicate everything, except for the *initial* block that controls the testcase. The different testcases would be implemented by providing different control blocks. Everything else would remain the same. But replication is not reuse. It creates additional physical copies that have to be maintained. If you had to write fifty testbenches, you would have to maintain fifty copies of the same bus-functional models.

Move the testcase control into a higher-level of hierarchy.

The implementation of reusable verification components in Verilog is relatively simple. Leave the portions of the testbench that are the same for all testbenches in the level of hierarchy immediately surrounding the design under verification and move the control structure unique to each testcase into a higher level of hierarchy.

Instead of invoking the bus-functional models directly, they are invoked using a hierarchical name. The level of hierarchy containing the reusable verification components, called the *test harness,* provides the procedural interface to the design under verification.

Figure 6-3 illustrates the difference in testbench structure. Figure 6-3(a) shows the original, non-reusable structure, while Figure 6-3(b) shows the same testbench using the reusable test harness structure. Sample 6-3 shows the implementation of the testcase shown earlier in Sample 6-1, using a reusable test harness. Notice how the tasks are now invoked using a hierarchical name.

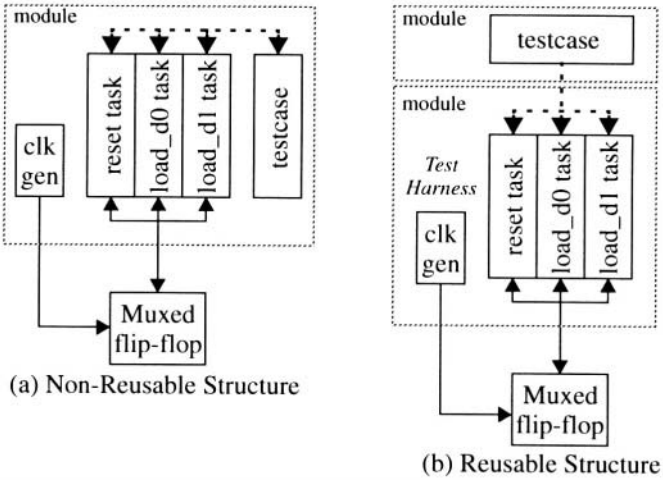


Figure 6-3. Verilog implementation of a testbench using a test harness structure

The test harness includes everything needed to operate the design.

The test harness should be self-contained and provide all signals necessary to properly operate the design under verification. In addition to all the low-level bus-functional models, it should include the clock and reset generators. The reset generator should be encapsulated in a task. This lets testcases trigger the reset operation at will, if required.

Packaging Bus-Functional Models

Bus-functional models can be reused between harnesses.

The structure shown in Figure 6-3 lets the test harness be reused between many testcases on the same design under verification. But it does not help the reusability of bus-functional models between test harnesses for different designs.

Sample 6-3. Using a reusable test harness in Verilog

```
module testcase;

harness th();

initial
begin: test_sequence
   th.sync_reset;
   th.load_d0(1'b1);
   ...
   $finish;
end
endmodule

module harness;

reg rst, d0, d1, sel, clk;
wire q, qb;

muxed_ff duv(d0, d1, sel, q, qb, clk, rst);

parameter cycle  = 100,
          Tsetup =  15,
          Thold  =   5;

always
begin
   #(cycle/2) clk = 1'b0;
   #(cycle/2) = 1'b1;
end

task sync_reset;
   ...
endtask

task load_d0;
   input data;
begin
   ...
end
endtask

task load_d1;
   ...
endtask
endmodule
```

All the tasks providing a complete bus-functional model for a given interface should be packaged to make them easy to reuse between test harnesses. For example, all of the tasks encapsulating the oper-

ations of a PCI bus should be packaged into a single PCI bus-functional model package to facilitate their reuse. Some of these tasks have been shown in Sample 5-61 and Sample 5-63.

Package a bus-functional model in its own level of hierarchy.

We made the test harness reusable by isolating it from its testcases in its own level of hierarchy. The testcases using the test harness simply have to instantiate it and use its procedural interface through hierarchical calls to use it. A similar strategy can be used to make bus-functional models reusable. All of the tasks encapsulating the operations are located in a module, creating a self-contained bus-functional model. For more details, see "Encapsulating Useful Subprograms" on page 94.

The procedural interface is accessed hierarchically and the physical interface is accessed through pins.

The signals driven or monitored by the tasks are passed to the bus-functional model through pins. The bus-functional model is instantiated in the test harness and its pins are properly connected to the design under verification. The tasks in the bus-functional model provide its procedural interface. They are called by the testcase using hierarchical names through the test harness.

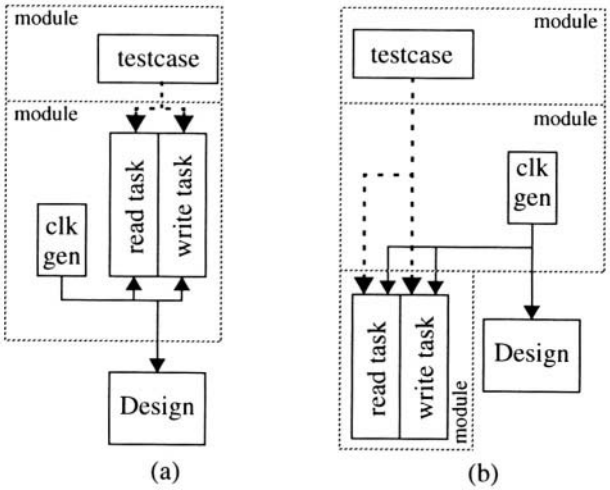


Figure 6-4. Test harness structures using an i386SX bus-functional model

Figure 6-4(a) shows the structure of a test harness with a non-reusable bus-functional model of an Intel 386SX processor. The bus-functional model is composed of two tasks: *read* and *write*

Figure 6-4(b) shows a functionally equivalent harness using a properly packaged bus-functional model. Sample 6-4 shows the skeleton Verilog code implementing an Intel 386SX bus-functional model, while Sample 6-5 and Sample 6-6 show how it can be used by the test harness and testcase, respectively.

Sample 6-4. Packaged bus-functional model for a i386SX

```
module i386sx(clk, addr, ads, rw, ready, data);
input          clk;
output [23:0]  addr;
output         ads ;
output         rw;
input          ready;
inout  [15:0]  data;

reg  [23:0]  addr;
...
reg [15:0] data_o;
assign data = data_o;

initial
begin
   ads    = 1'b1;
   data_o = 16'hZZZZ;
end

task read;
   ...
endtask

task write;
   ...
endtask
endraodule
```

Utility Packages

Mid-level utility routines are packaged in separate modules.

The utility routines that provide additional levels of abstraction to the testcases are also composed of a series of *tasks* and *functions.* They can be encapsulated in separate modules, using hierarchical names to access the lower-level procedural interfaces. The utility routines they provide would also be called using a hierarchical name.

Utility packages are never instantiated.

Because there is no *wire* or *register* connectivity involved between a utility package and the lower-level procedural interfaces, they need not be instantiated. They form additional simulation top-level

Sample 6-5. Test harness using the packaged i386SX bus-functional model

```
module harness;

reg       clk;
wire [23:0] addr;
...
wire [15:0] data;

i386sx cpu(clk, addr, ads, rw, ready, data);
design dut(clk, addr, ads, rw, ready, data, ...);

always
begin
   #50 clk = 1'b0;
   #50 clk = 1'b1;
end

task reset
begin
   disable cpu.read;
   disable cpu.write;
   ...
end
endtask

endmodule
```

Sample 6-6. Testcase using the packaged i386SX bus-functional model

```
module testcase;

harness th();

initial
begin: test_procedure
   reg [15:0] val;

   th.reset;
   th.cpu.read(24'h00_FFFF, val);
   val[0] = 1'b1;
   th.cpu.write(24'h00_FFFF, val);
   ...
end
endmodule
```

modules, running in parallel with the testbench and the design under verification. They can access the tasks and functions in the test harness using *absolute* hierarchical names.

Their own functions and tasks are also called using absolute hierarchical names. Sample 6-7 shows the implementation of a simple utility routine to send a fixed-length 64-byte packet, 16 bits at a time, via the i386SX bus using the Intel 386SX bus-functional model shown in Sample 6-4 and used in the test harness shown in Sample 6-5. Notice how an absolute hierarchical name is used to access the *write* task in the CPU bus-functional model in the harness in the testcase.

Sample 6-7. Utility package on test harness using packaged i386SX bus-functional model

```
module packet;

task send;
   input [64*8:1] pkt;
   reg   [ 15:0] word;
   integer i;
begin
   for ( i = 0 ; i < 3 2 ; i = i + 1) begin
      word = pkt[16:1];
      testcase.th.cpu.write(24'hl0_0000 + i,
                            word);
      pkt = pkt >> 16;
   end
end
endmodule
```

The harness is not intantiated either.

If the test harness is instantiated by the top-level testcase module, as shown in Sample 6-6, the name of the testcase module is part of any absolute hierarchical name. You can standardize on using a single predefined module name for all testcase modules and restrict them to a single level of hierarchy, with the test harness instantiated under a predefined instance name.

A better alternative is to leave the test harness uninstantiated, forming its own simulation top-level module. The testcases would simply use absolute hierarchical names instead of the relative hierarchical names to access tasks and functions in the test harness.

Sample 6-8 shows the testcase previously shown in Sample 6-6, but using absolute hierarchical names into an uninstantiated test harness. It also uses the packaged utility routine modified in Sample 6-9 to use the uninstantiated test harness. Figure 6-5 shows the structure of the simulation model, with the multiple top-levels.

Sample 6-8. Testcase using uninstantiated test harness

```
module testcase;

initial
begin: test_procedure
   reg [15:0] val;
   reg [64:8:1] msg;

   harness.reset;
   harness.cpu.read(24'h00_FFFF, val);
   val[0] = 1'b1;
   harness.cpu.write(24'h00_FFFF, val);
   ...
   packet.send(msg);
end
endmodule
```

Sample 6-9. Utility package on uninstantiated test harness

```
module packet;

task send;
   input [64*8:1] pkt;
   reg [15:0] word;
   integer i;
begin
   for (i = 0; i < 32; i = i + 1) begin
      word = pkt[16:1];
      harness.cpu.write(24'h10_0000 + i, word);
      pkt = pkt >> 16;
   end
end
endmodule
```

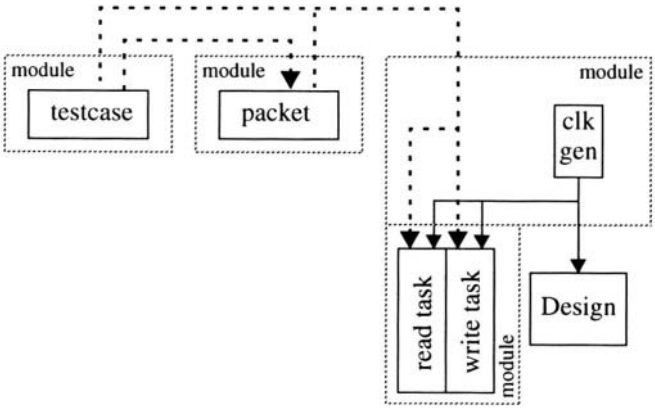


Figure 6-5. Simulation structure with uninstantiated harness and utility package

Additional top-levels are added to the command line.

It is very easy to create a Verilog simulation with multiple top-level modules. They are included in the simulation by simply adding their filename or module name to the simulation command. If you are using a simulator that compiles and elaborates the simulation structure in a single command, such as *Verilog-XL* or *VCS,* simply specify the additional filenames that compose the other top-levels. Assuming that the files involved in creating the structure shown in Figure 6-5, are named *testcase.v, packet.v, harness.v, i386sx.v,* and *design.v,* the command to use with *Verilog-XL* to simulate them would be:

```
% verilog testcase.v packet.v harness.v \
          i386sx.v design.v
```

For a simulation tool with separate compilation and elaboration phases, such as *ModelSim,* all of the required top-level modules must be identified to the simulation command:

```
% vlog testcase.v packet.v harness.v \
       i386sx.v design.v
% vsim testcase packet harness
```

As shown in Sample 6-10, the simulator displays the names of all the top-level modules in the simulation, and simulates them seamlessly, as one big model.

Sample 6-10. Simulator displaying top-level modules

```
...
The top-level modules are:
harness
testcase
packet

Simulation begins ...
...
```

Instantiating utility packages is too restrictive.

You might be tempted to require that all packages be instantiated one on top of each other. Lower-level utility package would instantiate the test harness, and higher-level packages would instantiate the lower-level packages. The structure of the testbench would thus follow the structure of the packages and only relative hierarchical names would be used. Unfortunately, the reverse would be occuring: the packages would be forced into following the structure of the testbench.

Follow a logical structure.

Requiring that every utility package be instantiated restricts their structure to a single hierarchical line. The test harness encapsulates the design and its surrounding bus-functional models into a single module. From that point on, only one module can be layered on top of it. It is not possible to create a tree in the hierarchy where all the branches terminate in the test harness.

Figure 6-6(a) shows the only possible structure allowed by instantiating all packages according to their abstraction layer. It is impossible to create the logical structure shown in Figure 6-6(b). The latter can be implemented using uninstantiated packages and absolute hierarchical names.

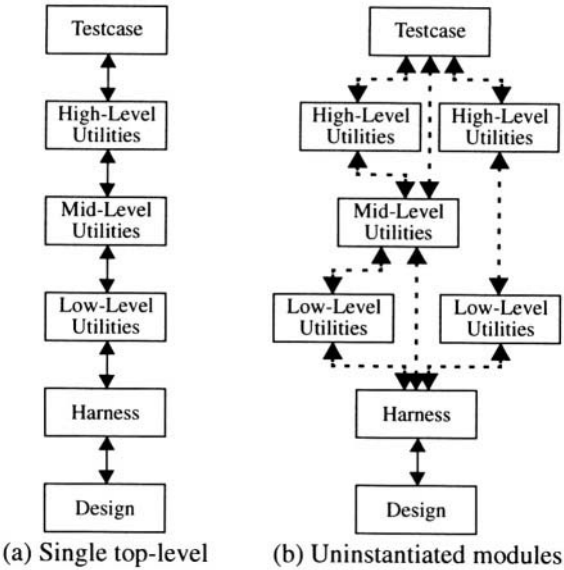


Figure 6-6. Simulation structures with harness and utility packages

Avoid cross-references in utility routines.

Because the utility packages are implemented in uninstantiated modules, they create a flat structure of globally visible procedural interfaces. The models do not enforce that they are used in a strictly layered fashion. It is possible - and tempting - to write utility packages that cross-reference themselves.

Sample 6-11 illustrates an example of cross-references, where packages use routines from each other. Cross-references make two packages inter-dependent. It is not possible to debug and verify one separately from the other. It also makes the packages more difficult

to reuse as they might have to be decoupled to be fitted into a different simulation environment.

When designing utility packages, stick to a strict set of layers. Packages can access utility routines in lower layers or within themselves, but never in a sibling package at the same level of abstraction. If a need for cross-references arises, question your design of the package set, or consider merging both packages into a single one.

Sample 6-11. Packages with cross-references

```
module syslog;

task note;
   input [80*8:1] msg;

   $write("NOTE: %0s\n", msg);
endtask

task terminate;
begin
   $write("Simulation terminated normally\n";
   watchdog.shutdown;
   $finish;
end
endtask

endmodule

module watchdog;
task shutdown;
begin
   syslog.note("Watchdog shutting down...");
   ...
end
endtask
endmodule
```

VHDL IMPLEMENTATION

This section evolves an implementation of the test harness and testbench architecture. Starting with a monolithic testbench, the implementation is refined into client/server bus-functional models, access and utility packages, and testcases. The goal is to obtain a flexible implemention strategy promoting the reusability of verifi-

cation components. This strategy can be used in most VHDL-based verification projects.

Creating another testcase simply requires changing the control block.

In Sample 6-2, the entire testbench is implemented in a single level of hierarchy. If you were to write another testcase for the muxed flip-flop design using the same bus-functional models, you would have to replicate everything, except for the body of the *process* that controls the testcase. The different testcases would be implemented by providing different sequential statements. Everything else would remain the same, including the procedures in the process declarative region. But replication is not reuse. It creates additional physical copies that have to be maintained. If you had to write fifty testbenches, you would have to maintain fifty copies of the same bus-functional models.

Packaging Bus-Functional Procedures

Bus-functional models can be located in a *package* to be reused.

One of the first steps to reducing the maintenance requirement is to move the bus-functional procedures from the process declarative regions to a *package*. These procedures can be used outside of the *package* by each testbench that requires them.

Sample 6-12. Bus-functional procedures for an i386SX

```
package i386sx is

subtype add_typ is std_logic_vector(23 downto 0);
subtype dat_typ is std_logic_vector(15 downto 0);

procedure read(        raddr: in    add_typ;
                       rdata: out   dat_typ;
                signal clk  : in    std_logic;
                signal addr : out   add_type;
                signal ads  : out   std_logic;
                signal rw   : out   std_logic;
                signal ready: in    std_logic;
                signal data : inout dat_typ);

procedure write(        waddr: in    add_typ;
                        wdata: in    dat_typ;
                 signal clk  : in    std_logic;
                 signal addr : out   add_type;
                 signal ads  : out   std_logic;
                 signal rw   : out   std_logic;
                 signal ready: in    std_logic;
                 signal data : inout dat_typ);
end i386sx;
```

They require *signal-class* arguments.

However, bus-functional procedures, once moved into a *package,* require that all driven and monitored signals be passed as *signal-class* arguments (see "Encapsulating Bus-Functional Models" on page 97 and Sample 4-16 on page 98). Sample 6-12 shows the *package* declaration of bus-functional model procedures for the Intel 386SX processor. Notice how all the signals for the processor bus are required as *signal-class* arguments in each procedure.

Bus-functional model procedures are cumbersome to use.

Sample 6-13 shows a process using the procedures declared in the package shown in Sample 6-12. They are very cumbersome to use as all the signals involved in the transaction must be passed to the bus-functional procedure. Furthermore, there would still be a lot of duplication across multiple testbenches. Each would have to declare all interface signals, instantiate the component for the design under verification, and properly connect the ports of the component to the interface signals. With today's ASIC and FPGA packages, the number of interface signals that need to be declared, then mapped, can easily number in the hundreds. If the interface of the design were to change, even minimally, all testbenches would need to be modified.

Sample 6-13. Using bus-functional procedures

```
use work.i386sx.all;
architecture test of bench is
   signal clk  : std_logic;
   signal addr : add_type;
   signal ads  : std_logic;
   signal rw   : std_logic;
   signal ready: std_logic;
   signal data : dat_typ;
begin

   duv: design port map (..., clk, addr, ads,
                         rw, ready, data, ...);

   testcase: process
      variable data: dat_typ;
   begin
      ...
      read(some_address, data,
           clk, addr, ads, rw, ready, data);
      ...
      write(some_other_address, some_data,
           clk, addr, ads, rw, ready, data);
      ...
   end process testcase;
end test;
```

Creating a Test Harness

The test harness contains declarations and functionality common to all testbenches.

To reduce the amount of duplicated information from testbench to testbench, you must factor out their common elements into a single structure that they will share. The common elements in all testbenches for a single design are:

- Declaration of the component
- Declaration of the interface signals
- Instantiation of the design under verification
- Mapping of interface signals to the ports of the design
- Mapping of interface signals to the signal-class arguments of bus-functional procedures.

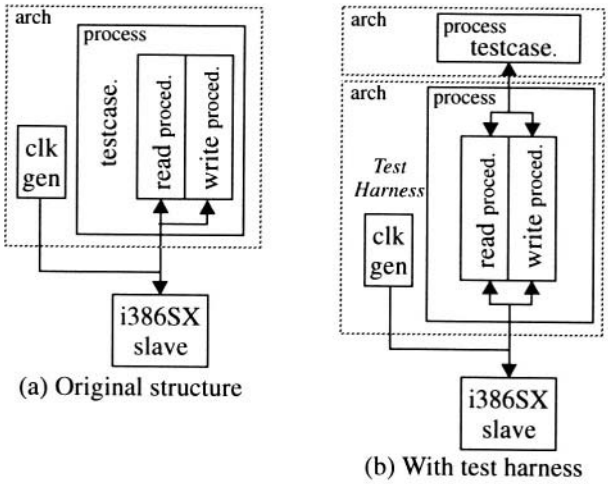


Figure 6-7. VHDL implementation of a testbench using a test harness structure

Use an intermediate level of hierarchy to encapsulate the test harness.

Figure 6-7(a) illustrates the structure of the testbench shown in Sample 6-13. The *read* and *write* procedures are shown in their invocation context, not their declaration context. Whether you use bus-functional procedures declared in the process declarative region, or in a package, the testbench structure remains the same. This is because the signal drivers are associated with the calling process.

Figure 6-7(b) illustrates how the same testbench can be structured to take advantage of the replicated functionality in multiple test-

benches. The replicated functionality is located in a lower-level architecture, called a *test harness.* The testcase drives the design under verification by instructing *processes* in the test harness to perform various operations.

A process in the harness owns the bus-functional procedures.

Because the signals that interface to the design under verification are local to the test harness, the bus-functional procedures must be called by a local process. The bus-functional procedures use the drivers associated with that local process. The testcase control process must instruct the local process, through control signals, to perform the appropriate cycle and return any relevant information.

This local process is often call a *server* process, while the testbench control process is called a *client* process. The control signals have to be visible to both the client and server processes, located in a different architecture. This can be accomplished in two ways:

- Passing them as ports on the test harness entity
- Making them global signals in a package.

Sample 6-14. Client/server control package

```
package i386sx is

type do_typ is  (read, write);

subtype add_typ is std_logic_vector(15 downto 0) ;
subtype dat_typ is std_logic_vector(15 downto 0) ;

type to_srv_typ is record
  do  : do_typ ;
  addr: add_type;
  data: dat_typ;
end record;

type frm_srv_typ is record
   data: dat_typ;
end record;

signal to_srv :  to_srv_typ;
signal frm_srv:  frm_srv_typ;

end i386sx;
```

Since a package is required to contain their type definitions, the latter does not require additional library units. Furthermore, using global signals eliminates the need for each testbench architecture to

declare them, then mapping them to the ports of the test harness. Sample 6-14 shows an implementation of the client/server control package for controlling the i386SX bus-functional procedures. The server process is in the test harness shown in Sample 6-15.

Sample 6-15. Server process in test harness

```
use work.i386sx.all;
architecture test of bench is
   signal clk  : std_logic;
   signal addr : add_type;
   signal ads  : std_logic;
   signal rw   : std_logic;
   signal ready: std_logic;
   signal data : dat_typ);
begin

   duv: design port map ( ... , clk, addr, ads,
                          rw, ready, data,  ...);

   i386sx_server: process
      variable data: dat_typ;
   begin
      wait on to_srv'transaction;
      if to_srv.do = read then
         read(to_srv.addr, data,
              clk, addr, ads, rw, ready, data);
      elsif to_srv.do = write then
         write(to_srv.addr, to_srv.data,
              clk, addr, ads, rw, ready, data);
      end if;
      frm_srv.data <= data;
   end process i386sx_server;
end test;
```

Use '*transaction* to synchronize operations.

Notice how the server process is sensitive to transactions on the *to_srv* control signal. This way, it is triggered after every assignment to the control signal, whether they are the same or not. Had the process been sensitive to events on *to_srv,* the second of two consecutive identical operations, as shown in Sample 6-16, would be missed.

Records are used to implement the control signals.

User-defined record types are used for the client/server control signals. Even if the record contains a single element, such as the *frm_srv_typ* record in Sample 6-14. A record is used to minimize maintenance if the control protocol between the client and the server needs to be modified. Fields can be removed, added or modified without affecting the type declaration of the control signals

Sample 6-16. Performing two identical operations back-to-back

```
to_srv <=  (do   => write,
            addr => (others => '1'),
            data => (others => '0.'));
wait on frm_srv.data'transaction;
to_srv <=  (do   => write,
            addr => (others => '1'),
            data => (others => '0'));
```

themselves, minimizing the impact on clients and server processes using them.

Abstracting the Client/Server Protocol

The client must properly operate (he control signals to the server process.

Sample 6-17 shows a client process accessing the services provided by the i386SX server process in the test harness shown in Sample 6-15. Notice how the client process waits for a transaction on the return signal to detect the end of the operation. This behavior detects the end of an operation that produces the same result as the previous one. If the client process had been sensitive to events on the return *frm_srv* signal, the end of the operation could have been detected only if it produced a different result from the previous one.

Sample 6-17. Client process controlling the server in test harness

```
use work.i386sx.all;
architecture test of bench is
begin

   i386sx_client: process
      variable data: dat_typ;
   begin
     ...
     --  Perform a read
      to_srv.do   <= read;
      to_srv.addr <= . . . ;
      wait on frm_srv'transaction;
      data := frm_srv.data;
      ...
      -- Perform a write
      to_srv.do   <= write;
      to_srv.addr <= ...;
      to_srv.data <= ...;
      wait on frm_srv'transaction;
      ...
   end process  i386sx_client;
end test;
```

Encapsulate the client/server operations in procedures.

Defining a communication procotol on signals between the client and the server processes does not seem to accomplish anything. Instead of having to deal with a physical interface documented in the design specification, we have to deal with an arbitrary protocol with no specification. Just as the operation on the physical interface can be encapsulated, the operations between the client and server can also be encapsulated in procedures. This encapsulation removes the client process from knowing the details of the protocol with the server. The protocol can be modified without affecting the testcases using it through the procedures encapsulating the operations.

Sample6-18. Client/server access package

```
package i386sx is

type do_typ is (read, write);

subtype add_typ is std_logic_vector(15 downto 0);
subtype dat_typ is std_logic_vector(15 downto 0);

type to_srv_typ is record
  do  : do_typ;
  addr: add_type;
  data: dat_typ;
end record;

type frm_srv is record
   data: dat_typ;
end record;

procedure read(       addr    : in  add_typ;
                      data    : out dat_typ;
                signal to_srv : out to_srv_typ;
                signal frm_srv: in  frm_srv_typ);

procedure    write(    addr    : in  add_typ;
                       data    : in  dat_typ;
                 signal to_srv : out to_srv_typ;
                 signal frm_srv: in  frm_srv_typ);

signal to_srv : to_srv_typ;
signal frm_srv: frm_srv_typ;

end i386sx;
```

Put the server access procedure in the control package.

The server access procedures should be located in the package containing the type definition and signal declarations. Their implementation is closely tied to these control signals and should be located with them. Sample 6-18 shows how the *read* and *write* access pro-

cedures would be added to the package previously shown in Sample 6-14.

Client processes use the server access procedures.

The client processes are now free from knowing the details of the protocol between the client and the server. To perform an operation, they simply need to use the appropriate access procedure. The pair of control signals to and from the server must be passed to the access procedure to be properly driven and monitored. Sample 6-19 shows how the client process, originally shown in Sample 6-17, is now oblivious to the client/server procotol.

Sample 6-19. Client process using server access procedures

```
use work.i386sx.all;
architecture test of bench is
begin

   i386sx_client: process
      variable data: dat_typ;
   begin
      ...
      -- Perform a read
      read(..., data, to_srv, frm_srv);
      ...
      -- Perform a write
      write(..., ..., to_srv, frm_srv);
      ...
   end process i386sx_client;
end test;
```

The testcase must still pass signals to and from the bus-functional access procedures. So, what has been gained from the starting point shown in Sample 6-13? The answer is: a lot.

Testbenches are now removed from the physical details.

First, the testcase need not declare all of the interface signals to the design under verification, nor instantiate and connect the design. These signals can number in the high hundreds, so a significant amount of work duplication has been eliminated.

Second, no matter how many signals are involved in the physical interface, you need only pass two signals to the bus-functional access procedures. the testcases are completely removed from the physical interface of the design under verification. Pins can be added or removed and polarities can be modified without affecting the existing testcases.

Managing Control Signals

Use separate unresolved *to* and *from* control signals.

Using two control signals, one to send control information and synchronization to the server, and vice-versa is the simplest solution. The alternative is to use a single signal, where both client and server processes each have a driver. A resolution function would be required, including a mechanism for differentiating between the value driven from the server and the one driven from the client.

You could use a single resolved control signal.

If you want to simplify the usage of the access procedures and the syntax of the client processes, a single resolved control signal can be used between the client and server processes. Instead of having to pass two signals to every server access procedures, only one signal needs to be passed. The price is additional development effort for the server access package - but since it is done only once for the entire design, it may be worth it.

But the risks outweigh the benefits.

Inserting a resolution function between the client and the server also introduces an additional level of complexity. It can make debugging the client/server protocol and the testcases that use it more tedious. It also makes it possible to have separate processes drive the control signal to the server process. Because that signal is now resolved, no error would be generated because of the multiple driver. Without proper interlocking of the parallel requests to the server, this would create a situation similar to Verilog's non-reentrant tasks.

Use qualified names for access procedures.

In a test harness for a real design, there may be a dozen server processes, each with their own access package and procedures. A real-life client process, creating a complex testcase, uses all of them. It may be difficult to ensure that all identifiers are unique across all access packages. In fact, making identifier uniqueness a requirement would place an undue burden on the authoring and reusability of these packages.

The identifier collision problem can be eliminated by using qualified names when using access procedures. Sample 6-20 shows a testbench using qualified names to access the *read* procedure out of the *i386sx* package. Notice how the *use* statement for the package does not specify "*.all"* to make all of the identifiers it contains visible.

Sample 6-20. Client process using qualified identifiers

```
use work.i386sx;
architecture test of bench is
begin
   i386sx_client: process
      variable data: i386sx.dat_typ;
   begin
      ...
      -- Perform a read
      i386sx.read(..., data, i386sx.to_srv,
                  i386sx.frm_srv);
      ...
   end i386sx_client;
end test;
```

Multiple Server Instances

Provide an array of control signals for multiple instances of the same server processes.

Designs often have multiple instances of identical interfaces. For example, a packet switch design would have multiple packet input and output ports, all using the same physical protocol. Each can be stimulated or monitored using separate server processes using the same bus-functional procedures. The clients needs to have a way to identify which server process instance they want to operate on to perform operations on the proper interface on the design.

Using an array of control signals, one pair for each server, meets this requirement. Sample 6-21 shows the access package containing an array of control signals, while Sample 6-22 shows one instance of a server process.

Sample 6-21. Array of client/server control signals for multiple servers

```
package i386sx is
...

type to_srv_ary_typ  is array(integer range <>)
   of to_srv_typ;
type frm_srv_ary_typ is array(integer range <>)
   of frm_srv_typ;

signal to_srv  : to_srv_ary_typ (0 to 7);
signal frm_srv : frm_srv_ary_typ(0 to 7);

end i386sx;
```

Sample 6-22. One instance of server process using an array of control signals

```
use work.i386SX.all;
architecture test of bench is
   ...
begin

...

   i386sx_server: process
      variable data: dat_typ;
   begin
      wait on to_srv(3)' transaction;
      if to_srv(3).do = read then
         read(to_srv(3).addr, data, ...);
      elsif to_srv(3).do = write then
         write(to_srv(3).addr, to_srv(3).wdat,
               ...);
      end if;
      frm_srv(3).rdat <= (others => 'X');
   end process i386sx_server;
end test;
```

You may be able to use the *far-generate* statement.

If the physical signals for the multiple instances of a port are properly declared using arrays, a *far-generate* statement can be used to automatically replicate the server process. Sample 6-23 illustrates this.

Sample 6-23. Generating multiple instances of a server process

```
use work.rs232.all;
architecture test of bench is
   signal tx: std_logic_vector(0 to 7);

begin

   duv: design port map(tx0 => tx(0),
                        tx1 => tx(1), ...);

   servers: for I in tx'range generate
      process
         variable data: dat_typ;
      begin
         wait on to_srv(I)'transaction;
         receive(data, tx(I));
         frm_srv(I).rdat <= data;
      end process;
   end generate servers;
end test;
```

Testbench generation tools can help in creating the test harness and access packages.

If you are discouraged by the amount of work required to implement a VHDL test harness and access packages, remember that it will be the most leveraged verification code. It will be used by all testbenches so investing in implementing a test harness that is easy to use returns the additional effort many times. Testbench generation tools, such as *Quickbench* by Chronology, can automate the generation of the test harness from a graphical specification of the timing diagrams describing the interface operations.

Utility Packages

Utility routines are packaged in separately.

The utility routines that provide additional levels of abstraction to the testcases are also composed of a series of *procedures.* They can be encapsulated in separate packages using the lower-level access packages. Sample 6-24 shows the implementation of a simple utility routine to send a fixed-length 64-byte packet, 16 bits at a time, via thei386SX bus using the Intel 386SX access package shown in Sample 6-18. Sample 6-25 shows a testcase using the utility package defined in Sample 6-24.

Sample 6-24. Utility package using i386SX bus-functional model access package

```
use work.i386sx.all;
package packet is

type packet_typ is array(integer range <>)
   of std_logic_vector(15 downto 0);

procedure send(       pkt     : in  packet_typ;
                signal to_srv : out to_srv_typ;
                signal frm_srv: in  frm_srv_typ);

end packet;

package body packet is

procedure send(       pkt     : in  packet_typ;
                signal to_srv : out to_srv_typ;
                 signal frm_srv: in  frm_srv_typ)
is
begin
   for I in pkt'range loop
      write(..., pkt(I), to_srv, frm_srv);
   end loop;
end send;
end packet;
```

Sample 6-25. Testcase using utility procedure

```
use work.i386sx;
use work.packet;
architecture test of bench is
begin

   testcase: process
      variable pkt: packet_typ(0 to 31);
   begin
      ...
      -- Send a packet on i386 i/f
      packet.send(pkt, i386SX.to_srv,
                       i386SX.frm_srv);
      ...

   end process testcase;
end test;
```

AUTONOMOUS GENERATION AND MONITORING

This section explains how properly packaged bus-functional models can become active entities. They can remove the testcase from the tedious task of generating background or random data, or performing detailed response checking.

Packaged bus-functional models create an opportunity.

Once the bus-functional procedures are moved in a module or controlled by an entity/architecture independent from the testcase, it creates an opportunity to move the tedious housekeeping tasks associated with using these bus-functional models along with them.

The packaged bus-functional models can now contain *processes* and *always* or *initial* blocks. These concurrent behavioral descriptions can perform a variety of tasks such as safety checks, data generation, or collecting responses for later retrieval. Instead of requiring constant attention by the testcase control process, these packaged bus-functional models could instead be configured to autonomously generate data according to configurable parameters. They could also be configured to monitor output response, looking for unusual patterns or specific data, and notifying the testbenches only when exceptions occur.

Autonomous Stimulus

Protocols may require more data than is relevant for the testcase.

Imagine a protocol on a physical interface that requires data to be continuously flowing. This procotol would obviously include a "data not valid" indication.

Figure 6-8 shows such an interface, where ATM cells are constantly flowing, aligned with a cell boundary marker. If this interface were to be stimulated using procedures only, the testcase would have to implement a control structure to continuously call the *send_cell* procedure not to violate the protocol. Most of the time, an invalid or predefined cell would be sent. But under further control by the testcase, a valid cell, relevant to the testcase, would be appropriately inserted. This control structure will likely have to be repeated in all testbenches generating input for that interface.

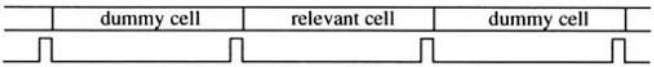


Figure 6-8. ATM protocol requiring continuous data flow

The access procedures would interface with a transmission process.

The *send_cell* procedure, contrary to previous implementations, would not immediately cause a cell to be sent on the physical interface and return once it has been transmitted. Instead, it would synchronize with the process transmitting dummy cells to have the relevant cell inserted in the data stream at the appropriate point.

It could also provide blocking or non-blocking implementations. In a blocking implementation, the *send_cell* procedure would return only when the cell was transmitted. In a non-blocking implementation, the *send_cell* procedure would return immediately, queueing the cell for future transmission.

Sample 6-26 shows an implementation of a blocking *send_cell* procedure in Verilog, while Sample 6-27 and Sample 6-28 show a non-blocking implementation in VHDL. Both blocking or non-blocking

implementations could be provided and selected using an additional argument to the procedure.

Sample 6-26. Blocking access procedure in bus-functional model

```
module atm_src(...);
...

task xmit_cell;
   input [ . . . ] cell;
begin
   ...
end
endtask

reg blocked;
task send_cell;
   input [...] cell;
begin
   blocked = 1'bl;
   wait blocked === 1'b0;
end
endtask

reg [...] dummy_cell;
always
begin
   if (blocked === 1'bl) begin
      xmit_cell(send_cell.cell);
      blocked = 1'b0;
   end else begin
      xmit_cell(dummy_cell);
   end
end
endmodule
```

Sample 6-27. Non-blocking access procedure

```
package body atm_gen is

procedure    send_ce11(    cell:  in  atm_cell;
                    signal to_srv :  out to_srv_typ;
                    signal frm_srv:  in  frm_srv_typ)
is
   to_srv.cell <= cell;
   wait  on  frm_srv'transaction;
end send_cell;

end atm_gen;
```

Sample 6-28. Server processes supporting non-blocking access procedure

```
use work.atm_gen.all;
architecture test of bench is
   subtype queue_idx is integer range 0 to 99;
   type atm_cell_array is array(queue_idx)
      of atm_cell;
   signal cell_queue: atm_cell_array;
   signal tail      : queue_idx;
begin

   non_block_srv: process
   begin
      wait on to_srv'transaction;
      cell_queue(tail) <= to_srv.cell;
      tail <= (tail + 1) rem cell_queue'length;
      frm_srv.done <= not frm_srv.done;
   end process non_block_srv;

   apply_cell: process
      variable head: queue_idx;
   begin
      if head = tail then
         wait until head /= tail;
      end if;
      ...
      head := (head + 1) rem cell_queue'length;
   end process apply_cell;

end test;
```

Random Stimulus

The content of generated data can be random.

It is a small step from automatically repeating the same stimulus to generating random stimulus. Instead of applying invalid or predefined data values or sequences, the packaged bus model can contain an algorithm to generate random data, in a random sequence, at random intervals.

Sample 6-29 shows the implementation of the i386SX bus-functional model from Sample 6-4 modified to randomly generate read and write cycles at random intervals within a specified address

range. The *always* block could be easily modified to be interrupted by requests from the testcase to issue a specific read or write cycle on demand.

Sample 6-29. Bus-functional model for a i386SX generating random cycles

```
module i386sx(...);

task read;
   ...
endtask

task write;
   ...
endtask

always
begin: random_generator
   reg [23:0] addr;
   reg [15:0] data;

   // Random interval (0-255)
   #($random » 24);

   // Random even address
   addr[23:21] = 3'b000;
   addr[20: 1] = $random;
   addr[    0] = 1'b0;

   // Random read or write
   if ($random % 2) begin
      // Write random data
      write(addr, $random);
   end else begin
      // Read from random address
      read(addr, data);
   end
end
endmodule
```

Autonomous generators can help compute the expected response.

If the autonomous generators are given enough information, they may be able to help in the output verification. For example, the strategy used to verify the packet router illustrated in Figure 5-31 requires that a description of the destination be written in the payload of each packet using the format illustrated in Figure 5-32. The header and filler information could be randomly generated, but the description of the expected destination is a function of the randomly generated header.

Similarly, the CRC is computed based on the randomly generated payload and the destination descriptor. Sample 6-30 shows how such a generator could be implemented. The procedural interface (not shown) would be used to start and stop the generator, as well as filling the routing information in *rt_table*.

Sample 6-30. Random generator helping to verify output

```
always
begin: monitor
   reg `packet_typ pkt;

   // Generate the header
   pkt`src_addr = my_id;
   pkt`dst_addr = $random;
   // Which port does this packet goes to?
   pkt`out_port_id = rt_table[pkt`dst_addr];
   // Next in a random stream
   pkt`strm_id = {$random, my_id};
   pkt`seq_num = seq_num[pkt`strm_id];
   // Fill the payload
   pkt`filler = $random;
   pkt`crc = computer_crc(pkt);
   // Send the packet
   send_pkt(pkt);

   seq_num[pkt`strm_id] =
      seq_num[pkt`strm_id] + 1;
end
```

Injecting Errors

Generators can be configured to inject errors.

If the design is supposed to detect errors in the interface protocol, you must be able to generate errors to ensure that they are properly detected. An autonomous stimulus generator could randomly generate errors. It could also have a procedural interface to inject specific errors at specific points in time. Sample 6-31 shows the generator from Sample 6-30 modified to corrupt the CRC on one percent of the packets. To make sure they are properly dropped, the sequence number is not incremented for corrupted packets.

Autonomous Monitoring

Monitors must always be listening.

In Chapter 5, we differentiated between a generator and a monitor based on who is in control of the timing. If the testbench controls when and if an operation occurs, then a generator is used. If the design under verification controls the timing and if an operation

Sample 6-31. Randomly injecting errors

```
always
begin: monitor
   pkt`crc = computer_crc(pkt);

   ...
   // Randomly corrupt the CRC for 1% of cells
   if ($random %100 == 0) begin
      pkt`seq_num = $random;
      pkt`crc = pkt`crc ^ (1<<($random % 8));
   end else begin
      seq_num[pkt`strm_id] =
         seq_num[pkt`strm_id] + 1;
   end

   // Send the packet
   send_pkt(pkt);
end
```

occurs, then a monitor is used. In the latter situation, the testbench must be continuously listening to potential operations. Otherwise, activity on the output signals can be missed.

Figure 6-9 illustrates the timing of the behavior required by the testbench. The monitoring bus-functional procedures must be continuously called, with zero-delay between the time they return and the time they are called back. If there are gaps between invocation of the monitoring procedures, as shown in Figure 6-10, some valid output may be missed.

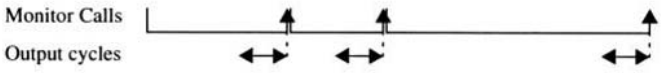


Figure 6-9. Proper call timing for monitoring procedures

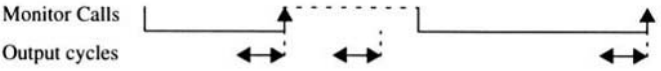


Figure 6-10. Delays between calls to monitoring procedures

Usage errors can be detected.

The role of a testbench is to detect as many errors as possible. Even from within the testbench itself. As the author of the test harness, your responsibility is not only to detect errors coming from the

design under verification, but also to detect errors coming from the testcase. If an unexpected output operation goes unnoticed because the testcase was busy checking the response from the previous cycle, it creates a crack that a bad design can slip through.

An autonomous output monitor can verify that the monitoring procedures are properly used. If activity is noticed on the output interface and the testcase is not actively listening, an error is reported. Sample 6-32 shows how an RS-232 monitor can be modified to detect if a serial transmission is started by the design before the testcase is ready to receive it.

Sample 6-32. Detecting usage errors in a RS-232 monitor.

```
process
begin
   loop
      -- Wait for testcase or serial Tx
      wait on to_srv'transaction, Tx;
      exit when not Tx'event;
      assert FALSE
         report "Missed activity on Tx"
         severity ERROR;
   end loop;
   -- Wait for the serial Tx to start
   wait on Tx;
   ...
end process;
```

Response can be collected for later retrieval.

Having to continuously call the output monitoring procedures can be cumbersome for the testcase. If the exact timing of output operations is not significant, only their relative order and data carried, the response can be autonomously monitored at all times. The data carried by each operation and an optional description of the operation are queued for later reception by the testcase.

The testcase then is free to introduce delays between the calls to the procedure that retrieves the next response. This procedure returns immediately if a previously received response is available in the queue. It may also block waiting for the next response if the queue is empty. A non-blocking option may be provided, along with a mechanism for reporting that no responses were available. Sample 6-33 shows how the server process for an RS-232 monitor returns responses from a queue implemented using an array. Sample 6-34 shows how the actual monitoring is performed in another process

that then puts the received data into the queue. Queues can be implemented using lists, as shown in "Lists" on page 115.

Sample 6-33. Server process in an autonomous RS-232 monitor.

```
process
begin
   wait on to_srv'transaction;
   -- Is queue empty?
   if pop = push then
      wait on push;
   end if;
   frm_srv.data <= queue(pop);
   pop <= (pop + 1) rem queue'length;
end process;
```

Sample 6-34. Monitor process in an autonomous RS-232 monitor.

```
process
begin
   wait until Tx = '1';
   ...
   — Is the queue full?
   assert (push+1) rem queue'length /= pop;
   queue(push) <= data;
   push <= (push + 1) rem queue'length;
end process;
```

Autonomous Error Detection

Data may contain a description of the expected response.

In "Packet Processors" on page 215, I described a verification strategy where the data sent through a design carried the information necessary to determine if the response was correct. Autonomous monitors can use this information to detect functional errors. Sample 5-68 shows an example of a self-checking autonomous monitor. The procedural interface of these monitors could provide configuration options to define the signature to look for in the received data stream.

INPUT AND OUTPUT PATHS

Each testcase must provide different stimulus and expect different responses. These differences are created by configuring the test harness in various ways and in providing difference data sequences. This section describes how data can be obtained from external files. It also shows how to properly configure reusable verification com-

ponents and how to ensure that simulation results are not clobbered by using unique output file names.

Programmable Testbenches

Data was assumed to be hardcoded in each testcase.

Throughout this chapter, there is no mention of the source of data applied as stimulus to the design under verification. Neither is there any mention of the source of expected response for the monitors. In most cases, the stimulus data and the expected response are specified in the verification plan and are hardcoded in the testcase. From the testcase, they are applied or received using bus-functional models in the test harness.

Testbenches can be programmed through external files.

A testcase can be implemented to read data to be applied to, or be expected from, the design under verification from external files. The external files can provide software instructions, a sequence of packets, video images, or sampled data from a previous design. It is a common strategy when the expected response is provided by an external C model, such as is illustrated in Figure 6-11. Programmable testbenches have an advantage: they do not need to be recompiled to execute a new testcase. When compilation or initialization times become critical factors, such as when SDF back-annotation is involved (see "SDF Back-Annotation" on page 305), they offer a technical solution to minimizing the number of time a model is compiled or initialized.

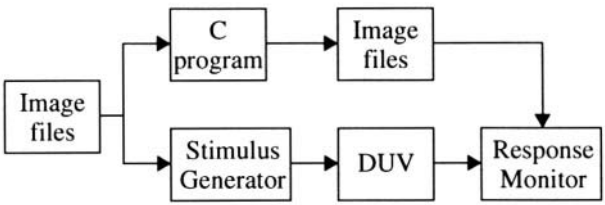


Figure 6-11. Using external input files for stimulus and response

Verilog and VHDL have file-input capabilities.

VHDL is capable of reading any text file, albeit in a very primitive fashion, using the general-purpose routines in the *textio* package. In Verilog, you can only read files of binary or hexadecimal values into a memory, using the *$readmemb* and *$readmemh* system tasks respectively. This requires that the external data representation first be compiled into binary or hexadecimal values before being read into a Verilog testbench. The testbench then interprets the numeric codes back into data or instructions.

Verilog's built-in file input tasks read the entire file into a memory. If the file contains large amount of sequential data, a large memory is required, consuming a significant portion of the available computing resources. Using a PLI function to read data in a sequential fashion is a better strategy. Only the required information is kept in memory during the simulation, improving performance. A link to an implementation of a *scanf*-like PLI task can be found in the *resources* section of:

`http://janick.bergeron.com/wtb`

Configuration Files

Configuration should be controlled by the testcase.

With the flexibility that packaged bus-functional models offer, they also offer the possibility of making testbenches difficult to understand or manage. Each bus-functional model should be self-contained, controlled through its procedural interface only. The functionality and implications of a testcase can be understood only by examining the top-level testcase control description. All of the non-default configuration settings originate from that single point.

Avoid using external configuration files.

The more files required to make a testcase complete, the more complicated the management task to reproduce a particular configuration. The complexity of file management grows exponentially with the number of files. External files should only be used for large data sequences, either for stimulus or comparison. Short files containing configuration information should be eliminated in favor of specifying the configuration in the testcase itself. Sample 6-35 shows an example of improper configuration using an external file. The random number generator should be seeded by the testcase control, using the task shown in Sample 6-36.

Sample 6-35. Improper configuration using an external file.

```
initial
begin: init_seed
   integer seed[0:0];

   $readmemh("seed.in", seed);
   $random(seed[0]);
end
```

Sample 6-36. Configuration using a procedural interface.

```
task seed;
   input [31:0] init;

   $random(init);
endtask
```

Make filenames configurable.

If you must use a file for reading input stimulus or data to compare against, do not hardcode the name of the file that must be used to provide this information in the bus-functional model. If a hardcoded pathname is used, it is not obvious, from the testcase control, that a file is used. If a filename must be specified through a procedural interface, it is immediately apparent that a file is used to execute a testcase. Sample 6-37 shows how a filename can be specified through a procedural interface in VHDL using the *string* type. The same flexibility can be provided in Verilog by simply allocating 8 bits per characters in a task input argument. Sample 6-38 shows an example and the section titled "Output File Management" on page 309 has more details.

Sample 6-37. User-specified filename in VHDL.

```
procedure data_from_file(name: in string) is
   file fp: text is in name;
begin
   ...
end data_from_file;
```

Sample 6-38. User-specified filename in Verilog

```
task data_from_file;
   input [8*32:1] name;

   $readmemh(name, mem);
endtask
```

Concurrent Simulations

Make sure filenames are unique.

There is another problem with using hardcoded pathnames. If multiple simulations must be run concurrently, a hardcoded filename creates collisions between two simulations. Each simulation tries to produce output to the same file, or read data from the same file. Each simulation must be able to run without conflicting with each other. Therefore, the filenames used for each testcase must be unique.

This problem is typically encountered when generating waveform trace files. By default, the trace information goes to a file with a

generic name, such as *verilog.dump* for VCD dump files. To guarantee that each testcase uses a different file, provide a user-specified filename that includes the name of the testcase. Sample 6-39 shows how a string *parameter* containing the testcase name in Verilog can be concatenated to a string literal to create a full filename. In VHDL, the concatenation operator would be used between two *string* expressions, as shown in Sample 6-40.

Sample 6-39. Generating unique filenames in Verilog.

```
parameter testcase = "...";

initial
begin
   $dumpfile({testcase, ".dump"});
   $dumpvars;
end
```

Sample 6-40. Generating uniquefilenames in VHDL.

```
architecture test of bench is
   constant testcase: string = "...";
begin
   process
   begin
      read_from_file(testcase &  ".dat");
      ...
   end process;
end test;
```

Compile-Time Configuration

Avoid using compile-time configuration of bus-functional models.

When a language offers a preprocessor, it is often used as the mechanism for configuring source code. A different configuration requires a recompilation of the source code using a different *header* file. With most Verilog simulators always recompiling the source code before each simulation, it is a technique that appears efficient and is easy to use. This technique should be discouraged to minimize the compilation requirements in compiled simulators or languages. It also makes managing a testcase more complicated as an additional file, separate from the testcase control, must be managed and kept up-to-date. Furthermore, it may be impossible to ensure the uniqueness of the header file name for each testcase configured by the preprocessor. Using compile-time configuration may make it impossible to run concurrent compiled simulations.

Mostcompile-time configurations are not modified for a specific testcase.

To minimize the number of files used in a testbench, a single compile-time configuration file is usually used. It contains the definitions for all configurable preprocessor symbols in the test harness. The majority of them have identical values from testcase to testcase, with only a few taking different testcase-specific values. Instead, providing a default value for the configuration parameters that do not need a specific value for a given testcase would avoid the needless duplication of information. Sample 6-41 shows an example of configuring the maximum time-out value of a watchdog timer using an external header file (shown in Sample 6-42) which defines the appropriate preprocessor definitions. Sample 6-43 shows how to provide the same configurability through a procedural interface. A sensible default value is provided that can be used by most testcases, requiring no specific configuration instructions.

Sample 6-41. Compile-time configuration.

```
module watchdog;
`include "defs.vh"

integer count;
initial count = 0;
always @ (posedge clk)
begin
   count = count + 1;
   if (count > `TIMEOUT) ...
end
endmodule
```

Sample 6-42. Compile-time configuration definition.

```
`define TIMEOUT     1000000
`define CLK_PERIOD 100
`define MAX_LENGTH 500
`define DUMPFILE     "testcase.dump"
```

Plan your configuration mechanism.

If different configurations must be verified with similar stimulus and response, consider architecting your testcase to facilitate its configuration from within the testbench. The next section should provides some useful techniques.

VERIFYING CONFIGURABLE DESIGNS

This section describes how to verify two kinds of design configurability: soft and hard.

Sample 6-43. Equivalent procedural configuration.

```
module watchdog;

integer count, max;
initial
begin
   count = 0;
   max   = 32'h7FFF_FFFF;
end

task timeout;
   input [31:0] val;

   max = val;
endtask

always @ (posedge clk)
begin
   count = count + 1;
   if (count > max) ...
end
endmodule
```

Soft configuration can be changed by a testcase.

There are two kinds of design configurability. The first is *soft* configurability. A soft configuration is performed through a programmable interface and can be changed during the operation of the design. Examples of soft configurations include the offsets for the almost-full and almost-empty flags on a FIFO, the baud rate of a UART, or the routing table in a packet router. Because it can be modified during the normal operation of a design, soft configuration parameters are usually verified by changing them in a testcase. Soft configuration is implicitely covered by the verification process.

Hard configuration cannot be changed once simulation starts.

The second kind of configurability is *hard* configuration. It is so fundamental to the functional nature of the design, that it cannot be modified during normal operations. For example, whether a PCI interface operates at 33 or 66 MHz is a hard configuration. So is the width and depth of a FIFO, or the number of master devices on an on-chip bus. Hard configuration parameters are constant for the duration of the simulation and often affect the testbench as well as the design under verification. A testbench must be properly designed to support hard configuration in a reproduceable fashion.

Configurable Testbenches

Configure the testbench to match the design.

If a design can be made configurable, so can a testbench. The configuration of the testbench must be consistent with the configuration of the design. Using a configuration technique similar to the one used by the design can help ensure this consistency. Using *generics* or *parameters* to configure the testbench and the design allows the configuration defined at the top-level to be propagated down the hierarchy, from the testcase control, through the test harness, and into the design under verification.

Parameters and generics can configure almost anything.

Generics and *parameters* were designed to create configurable models. They offer the capability to configure almost any declaration in a VHDL or Verilog testbench. You can use them to define the width of a data value, the length of an array, or the period of the clock signal. In VHDL, they can even be used to determine the number of instantiated devices using a *generate* statement. Wherever a constant literal value is used in a testbench, it can be replaced with a reference to a *generic* or a *parameter*.

Generics and parameters are part of a component interface.

If a testbench component is configurable using generics or parameters, they become part of its interface. Whenever a configurable component is used, the value of the generics or parameters must be specified, if they must differ from their default values.

In VHDL, this is accomplished using the *generic map* construct in an instantiation or configuration statement. In Verilog, it is done using the *defparam* statement or the *#()* construct in an instantiation statement.

Sample 6-44 shows the interface of a memory model with configurable numbers of address and data bits. To use this model in a board or system under verification, the *AWIDTH* and *DWIDTH* parameters must be properly configured. Sample 6-45 shows both methods available in Verilog (I prefer the first one because it is self-documenting and robust to changes in parameter declarations.) Sample 6-46 shows how to map generics in VHDL. Notice how the configurability of the system-level model is propagated to the memory model.

Sample 6-44. Configurable memory model.

```
module memory(data, addr, rw, cs);
parameter DWIDTH = 1,
          AWIDTH = 1;
   inout [DWIDTH-1:0] data;
   input [AWIDTH-1:0] addr;
   input rw;
   input cs;

endmodule
```

Sample 6-45. Using a configurable model in Verilog.

```
module system(...)
parameter ASIZE = 10,
          DSIZE = 16;
wire [DSIZE-1:0] data;
reg [ASIZE-1: 0] addr;
reg              rw, cs0, cs1;

memory m0(data, addr, rw, cs0) ;
defparam m0.AWIDTH = ASIZE,
         m0.DWIDTH = DSIZE;
memory #(DSIZE, ASIZE) m1(data, addr, rw, cs1);
endmodule
```

Sample 6-46. Using a configurable model in VHDL.

```
entity system is
   generic (ASIZE: natural := 10;
            DSIZE: natural := 16) ;
   port (...);
end system;

use work.blocks.all;
architecture version1 of system is
 signal data: std_logic_vector(DSIZE-1 downto 0) ;
 signal addr: std_logic_vector(ASIZE-1 downto 0);
 signal rw, cs: std_logic;
begin
 M0: memory generic map (DWIDTH => DSIZE,
                         AWIDTH => ASIZE)
            port map (data, addr, rw, cs);
end version1;
```

Top Level Generics and Parameters

Top-level modules and entities can have generics or parameters.

Each configurable testbench component has *generics* and *parameters,* defined by the higher-level module or architecture that instantiates them. Eventually, the top-level of the testbench is reached. The top-level module or entity in a design has no pins or ports, but

it can have parameters or generics. The top-level of a testbench can be configured using the same mechanisms as the lower-level components.

For example, Sample 6-47 shows the top-level module declaration for a testbench to verify a FIFO with a configurable width and depth. Notice how the module does not have any pins, but does have parameters.

Sample 6-47. Configurable top-level of a FIFO testbench.

```
module fifo_tb;
parameter WIDTH = 1,
          DEPTH = 1;
...
endmodule
```

Top-level generics or parameters need to be defined.

By definition, the top-level is not instantiated anywhere. It is the very top level of the simulation. How can its parameters or generics be set? Some simulation tools allow the setting of top-level generics or parameters via the command line. However, a command line cannot be archived. How then can a specific configuration be reproduced later or by someone else? Wrapping the command line into a script is one solution. But it may not be portable to a different simulator that does not offer setting top-level parameters or generics via the command line.

Use a *defparam* module in Verilog.

In Verilog, the top-level parameters can be set using a *defparam* statement and absolute hierarchical names. A configuration would be a module containing only a *defparam* statement, simulated as an additional top-level module. Sample 6-48 shows how the testbench shown in Sample 6-47 can be configured using a configuration module.

Sample 6-48. Configuration module for FIFO testbench.

```
module fifo_tb_config_0;
   defparam fifo_tb.WIDTH = 32,
            fifo_tb.DEPTH = 16;
endmodule
```

Use an additional level of hierarchy and configuration unit in VHDL.

In VHDL, the *configuration* unit does not allow setting top-level generics. To be able to set them, an additional level of hierarchy must be added. As Sample 6-49 shows, it is very simple and not specific to any testbenches since no ports or signals need to be

mapped. The configuration unit can then be used to configure the generics of the testbench top-level, as shown in Sample 6-50.

Sample 6-49. Additional level of hierarchy to set top-level generics.

```
entity config is
end config;
architecture toplevel of config is
   component testbench
   end component;
begin
   tb: testbench;
end toplevel;
```

Sample 6-50. Configuration unit for FIFO testbench.

```
configuration conf_0 of config is
for toplevel
   for tb use entity work.fifo_tb(a)
      port map(WIDTH = 32,
               DEPTH = 16);
   end for;
end for;
end conf_0;
```

SUMMARY

This chapter focused on the implementation of testbenches for a device under verification. It described an architecture that promotes reusing verification components. The portion of the testbenches that is common between all testcases is structured into a test harness. Each testcase is then implemented on top of the test harness, using a procedural interface to apply stimulus to and monitor response from the device under verification. Although external data files can be used, the configuration of bus-functional models by each testcases should be limited to using the available procedural interfaces.