Generating Test Stimulus

Module **22**
Revision 1.0
Version 27.0

Estimated Time:
- Lecture
- Lab

cadence

This module examines some common coding styles and methods applicable to testbench generation.

Module Objective

In this module, you:

- Generate test stimulus.

Topics

- Simulation process review
- Organizing the testbench
- Using hierarchical names
- Generating clocks
- Generating stimulus
  - "In-line" stimulus
  - Stimulus using loops
  - Stimulus using tasks
  - Random Stimulus
  - Applying Stimulus from a File
  - Testing "corner" conditions
  - Testing protocol interactions
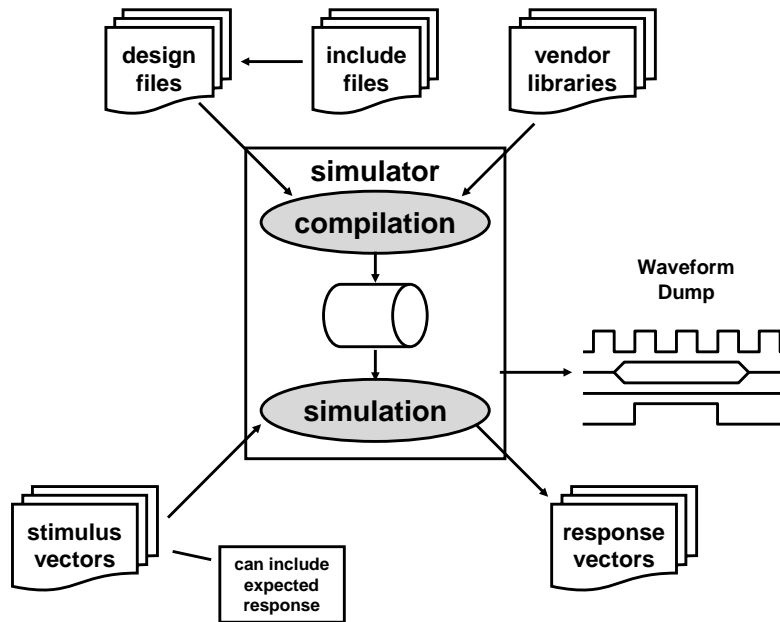 - Capturing and playing back vectors

cādence

Your objective is to generate test stimulus.

To do that, you need to know how to create a testbench and some common ways to generate stimulus.
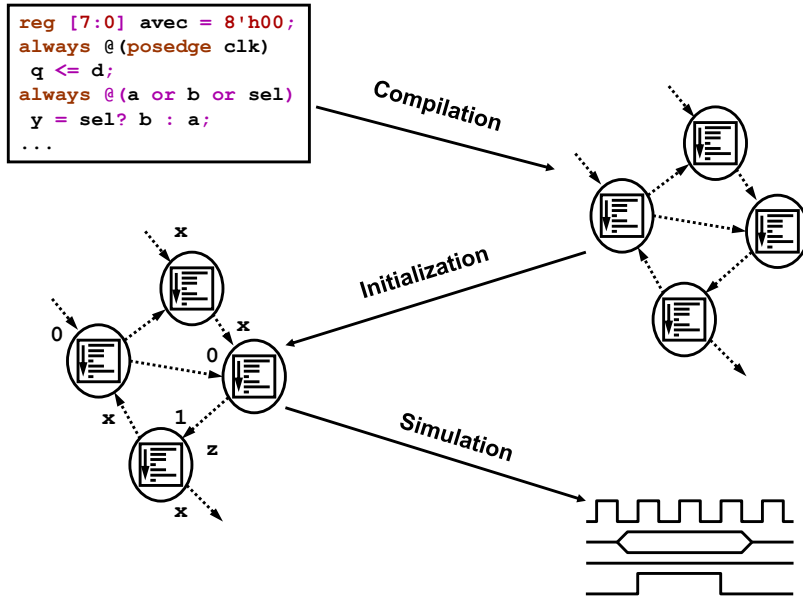
Simulation Inputs and Outputs Review

cādence

Input to the compiler is source text, which may reference third-party IP, and potentially other files to define the design configuration. Output from the compiler is a database representation of the design hierarchy.

Input to the simulator is of course the database representation of the design hierarchy, and can include stimulus vectors, may include expected response vectors, or perhaps some sort of pseudocode to drive an on-board stimulus generator. Output from the simulator is whatever diagnostic messages the user coded and value change data from whatever nets and variables the user probed.

429

Simulation Process Review

```
reg [7:0] avec = 8'h00;
always @(posedge clk)
 q <= d;
always @(a or b or sel)
 y = sel? b : a;
...
```

Compilation

Initialization

Simulation

cādence

Simulation of the model takes the following steps:

- The 1st step is to compile the model:

    The compiler parses the source text and constructs an internal database object for each design unit. A separate part of the compilation process, called elaboration, then links together the various parts of the design and builds a data structure representing the design hierarchy. For precompiled implementations, yet another process, called code generation, generates platform executable code for the design unit behaviors.

- The 2nd step is to initialize the elaborated data structure:

    The initialization phase initializes most nets to the high impedance state and initializes most variables to the unknown state. It initializes trireg nets to the unknown state and real variables to 0.

- The 3rd step is the actual simulation, starting at time 0:

    The simulation first propagates the design initialization events through the design hierarchy. The simulation then executes the statements in each initial and always construct up to the point at which a timing control blocks execution or the initial construct completes. Assignments in these statements can schedule events for the current and future times. As time advances, the simulator executes scheduled events, this execution often scheduling additional events. This process continues throughout the simulation.

The simulator initializes most variables to the unknown state. Variables remain at the unknown state until an assignment statement assigns some other state. A variable remaining in the unknown state throughout the simulation is an indication of a failure to assign a known state to the variable, most likely a failure to reset the design.

Most nets initialize to the high impedance state. A net remaining at the high impedance state throughout the simulation is an indication that nothing drives the net.

Organizing the Testbench

Simple testbench
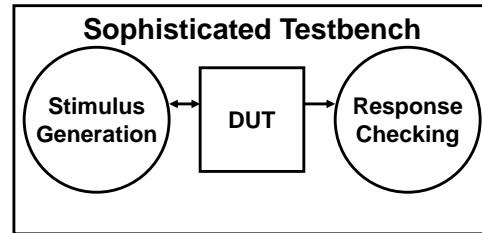
- Just sends data to design
- Few procedures
- No interaction
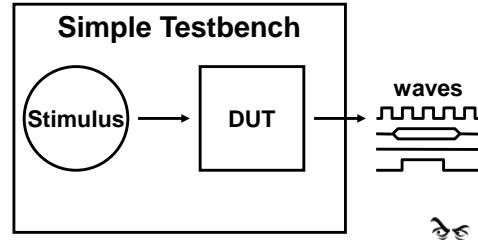
Sophisticated testbench

- Models system-level design environment
- Two-way communication
  - Stimulus generation
  - Response checking

We are most interested in sophisticated testbenches!

Simple Testbench

Stimulus → DUT → waves

Sophisticated Testbench

Stimulus Generation ↔ DUT → Response Checking

cādence

A simple testbench applies vectors to the design under test (DUT) and the user manually verifies the results.

Non-trivial projects use some form of "intelligent" or "smart" testbench that reacts to the DUT, e.g., for a bus protocol that requires some kind of handshake mechanism.

Such sophisticated testbenches are self-checking, i.e., they automatically verify the results.

Sophisticated testbenches typically instantiate modules for stimulus generation and response checking under a top-level "wrapper", to more easily swap different tests and design configurations "in" and "out".

Using Hierarchical Names

Every identifier has a unique hierarchical path name.

- Starts at a top module instance and downward traverses module instances, named blocks and tasks and functions.

- Each scope separated by "dot" (".").

- Hierarchical names can be used to access identifiers in other modules.
  - Also called Out Of Module References (OOMRs)

- There are restrictions:
  - Cannot access items of automatic tasks or functions
  - Cannot access items of unnamed blocks

- Hierarchical pathnames can be used anywhere but are primarily used in testbenches.

```verilog
module fsm (
 input clk, rst, do,
 output done);
 reg [2:0] state;
 // assume fsm behaviors
endmodule

module fsm_tb;
reg clk, rst, do; wire done;
fsm u1 (clk, rst, do, done);
initial begin
 $monitor($stime,,u1.state);
 // assume test behaviors
 end
endmodule
```

Path name can be:
- Downward relative to current scope.
- Downward starting at higher module or instance.

cādence

Every identifier has a unique hierarchical path name. The path name starts at a top module instance and traverses module instances, named blocks and tasks and functions, downward. Each such scope in the path name is separated from the next with the dot (.) character. You can make hierarchical references from anywhere in the design to almost anywhere else in the design. The exceptions are that you cannot make hierarchical references to items of automatic tasks and automatic functions, and you can make hierarchical references to items of an unnamed generate block only from within that generate block.

You can access items within the hierarchy rooted at the current scope by using relative path names. You access other items using absolute path names. The example testbench monitors the internal state of the FSM. This is a common use of hierarchical references – to gain testbench access to signals that are not available through module ports.

Hierarchical pathnames are primarily for testbench use, but can also be used in cross-hierarchy references, to access identifiers in one sub-module from another. As hierarchical pathnames cannot contain explicit "up-scope" references, cross-hierarchy references rely on the following resolution process:

- Hierarchical paths do NOT start at the top module instance.
- The paths are always relative to where they are used.
- If there is no match for the path downwards from the current scope, then the simulator automatically jumps up one scope and looks for a match from there.
- If there is still no match, then the simulator keeps jumping up until the top level module.
- If there is still no match from there, then the path is invalid and an elaboration error is produced.

If you use hierarchical names in testbenches only (as recommended) then pathname resolution is not a problem. The resolution is only an issue when you embed pathnames in hierarchical blocks, which we generally discourage (since it breaks reuse).

Generating Clocks

Behavioral Clocks

Beware

```
reg clk = 1;
always // simple
 #(PERIOD/2) clk = ~clk;
```

```
always @(clk)
 #5 clk = !clk;(doesn't oscillate)

always
clk <= #5 !clk;  (zero delay loop)
```

```
reg clk = 1;
initial // delayed
 #(DELAY) forever
  #(PERIOD/2) clk = ~clk;
```

```
reg clk = 1;
initial // irregular
 #(DELAY) forever
  begin
   #((  DUTY)*PERIOD) clk = 0;
   #((1-DUTY)*PERIOD) clk = 1;
  end
```

cadence

The master clock generator is seldom part of the design and many people place it in the top-level wrapper, because it is also not really part of the stimulus generation or response checking operations. For a typical large and complex design, efficiency of the clock generator itself is not usually a concern.

In the "Beware" example section, the coding style to generate clock does not oscillate the clock and can lock the simulator.

Generating Stimulus

This section examines stimulus generation:

- The incremental approach
- Generating in-line stimulus
- Generating stimulus using loops
- Generating stimulus using tasks
- Generating random stimulus
- Applying Stimulus from a File
- Testing "corner" conditions
- Testing protocol interactions
- Vector capture and playback

cadence

This section examines stimulus generation. This introduction to test generation gives you a pretty good idea of what you can do with Verilog to test your design. The SystemVerilog language, which is a superset of Verilog, has much more powerful features directly targeting test generation. Industry advocates provide the Universal Verification Methodology (UVM), based on SystemVerilog, and plug-and-play verification IP components with which you can quickly construct a sophisticated verification environment.

The Incremental Approach

One of your most valuable test strategies is to the incremental approach:

- Build incrementally on what you know works:
  - First test for signs of life
  - Then, test interface and basic functions
  - Finally, test internal complex functionality
- For example, test of a storage device might:
  - Test reset (if that feature exists)
  - Test data lines individually or in combinations
  - Test address lines individually or in combination
  - Test functional "corners" i.e., back-to-back and overlapping operations
  - Test timing "corners" i.e., frequency, setup/hold, recovery/removal

cādence

A test sequence may in general incrementally contain elements from the following:

- Check-in tests, to verify the model before checking it into source control.
- Connectivity tests of the model, ASIC or other subunit instantiation.
- Functional tests of major widely-used functional features.
- Deterministic worst-case tests on data and control.
- Random worst-case tests on data and control.
- Tests of asynchronous interactions, including clock/data margining.
- Tests of inter-ASIC datapath and data processing algorithms.
- Hardware subsystems test using basic system-level operations and protocols.
- System boot.
- System multitasking and exception handling.
- Randomly sequenced full-system tests with heavy workload.

Generating In-Line Stimulus

The simplest approach is to use in-line stimulus generation:

- You only need to specify the variable value transitions.

- You can easily specify complex timing relationships.

- This testbench can become very large for complex tests.

```verilog
module inline_tb;
 reg rd=0, wr=0;
 reg [4:0] addr;
 reg [7:0] dreg;
 wire [7:0] data=dreg;
 mem u1 (addr,data,rd,wr);
 initial
  begin
   #10 wr=1; addr=8'h00; dreg=8'h00;
   #10 wr=0;
   #10 rd=1; addr=8'h00; dreg=8'hZZ;
   #10 rd=0;
   ...
  end
endmodule
```

cadence

The simplest way to generate a stimulus is with an in-line sequence of assignments.

This works reasonably well for an individual test with an irregularly-timed control protocol, as you specify exactly the transitions you need and exactly the time that you need each one.

This testbench can become very large for complex tests. When you see this type of stimulus, it is almost always machine-generated stimulus derived from dumped test vectors.

This example does one memory write operation followed by one memory read operation.

Generating Stimulus Using Loops

For repeated stimulus using a regularly-timed control protocol, you can "roll" your inline stimulus into loops. Advantages include:

- Compact easier to understand and debug stimulus generation.

```verilog
module loop_tb;
 reg rd=0, wr=0;
 reg [4:0] addr;
 reg [7:0] dreg;
 wire [7:0] data=dreg;
 mem u1 (addr,data,rd,wr);
 initial
  begin: TEST
   integer i;
   for (i=0;i<=31;i=i+1)
    begin
     #10 wr=1; addr=i; dreg=i;
     #10 wr=0;
    end
   for (i=0;i<=31;i=i+1)
    begin
     #10 rd=1; addr=i; dreg=8'hZZ;
     #10 rd=0;
    end
  end
endmodule
```

cādence

For repeated stimulus using a regularly-timed control protocol, you can "roll" your inline stimulus into loops. This produces stimulus generation code that is more compact and easier to understand and debug.

This example uses loops to first fill a memory with data and then to examine the data in memory.

Even very sophisticated testbenches use loops to generate stimulus, but they also take advantage of other techniques that we will examine next.

Generating Stimulus Using Tasks

Use a task if:

- You need the same sequence of control signals more than once.
- Or even if you just want to encapsulate them.

```verilog
task write ( input [4:0] a,
             input [7:0] d );
 begin
  #10 wr=1; addr=a; dreg=d;
  #10 wr=0;
 end
endtask

task read ( input  [4:0] a,
            output [7:0] d );
 begin
  #10 rd=1; addr=a; dreg=8'hZZ;
  #10 rd=0; d=data;
 end
endtask

initial begin: TEST
  integer i;
  for (i=0;i<=31;i=i+1)
   // call tasks
 end
```

cadence

Placing operations in tasks enhances readability, and if the operations are required in more than one place, it is essential for compact and concise code.

This illustration encapsulates the write and read operations in tasks, which procedural statements anywhere in the design can call upon.

Generating Random Stimulus

$random [(*seed*)]

- Takes (and returns) a 32-bit seed
  - Seed the generator to produce repeatable sequences.
- Returns a random 32-bit signed integer
  - Verilog-1995: Algorithm is not standard and not portable.
  - Verilog-2001: Algorithm is standard and portable.

You can alter the random distribution:

- **$dist_chi_square(*seed*,*dgf*)**
- **$dist_erlang(*seed*,*k*,*mean*)**
- **$dist_exponential(*seed*,*mean*)**
- **$dist_normal(*seed*,*mean*,*std*)**
- **$dist_poisson(*seed*,*mean*)**
- **$dist_t(*seed*,*dgf*)**
- **$dist_uniform(*seed*,*start*,*end*)**

```verilog
module random_tb;
 reg rd=0, wr=0;
 reg  [4:0] addr;
 reg  [7:0] dreg,want,got;
 wire [7:0] data=dreg;
 mem u1 (addr,data,rd,wr);
 // assume task definitions
 initial begin: TEST
   integer i, seed;
   seed = 1;
   for (i=0;i<=31;i=i+1)
    write(i,$random(seed));
   seed = 1;
   for (i=0;i<=31;i=i+1) begin
     read(i,got);
     want = $random(seed);
     if (got !== want)
      // handle exception
   end
 end
endmodule
```

cādence

The operating environment for many devices includes some random inputs. A good test suite includes tests that mimic this randomness as much as practical. The random data might find errors in a design that more predictable data does not.

You can use the $random system function to easily create random data patterns.

- The function takes and returns an integer seed, so the seed must be a variable and not just a value. You can provide an initial seed value to produce a repeatable sequence of pseudo-random numbers, and you can provide each function call instance with its own seed to make the function call instance independent of other function call instances.

- The function returns integer values. For random numbers wider than an integer, you can concatenate multiple $random calls.

The $random system function returns numbers uniformly distributed within the range of an integer. Realistically modeling the randomly distributed environment values usually requires a more complex distribution. Verilog provides a set of the most classic distributions that will probably be sufficient for your needs.

This example in a loop calls the write task and passes random data to write, then in a loop calls the read task to get the memory data and check it. Before each loop the example initializes the seed so that the random number sequences are the same for both loops.

Note: Simulators compliant with only the Verilog 1995 standard can produce different random number sequences. Simulators compliant with the Verilog 2001 standard will produce identical random number sequences.

Common Random Distributions
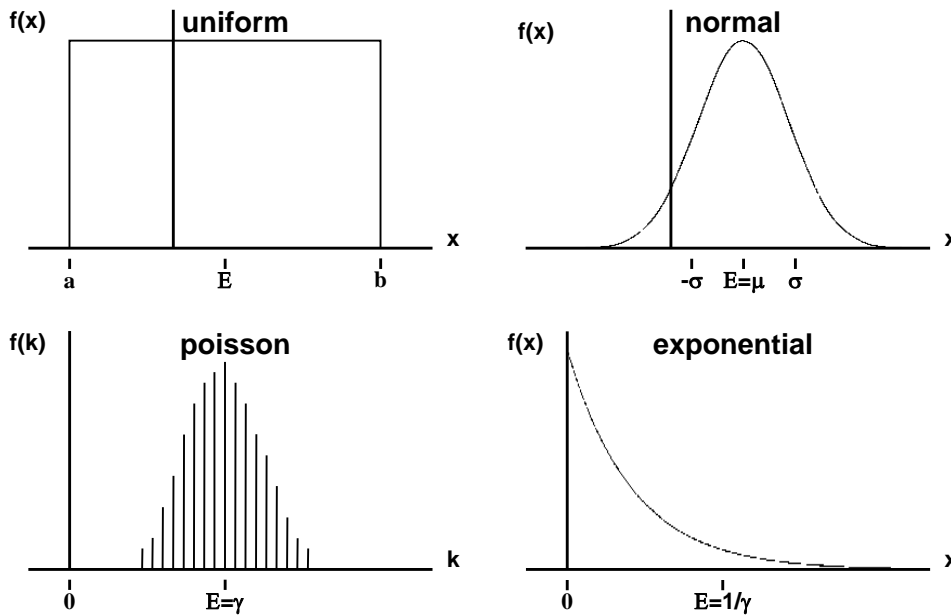
cādence

Verilog provides a selection of commonly used distribution functions. These are the four that you are most likely to use in a testbench:

- The uniform distribution equally distributes values between two limits. You use this distribution for a data item that is equally likely to be any value between the two limits.

- The normal distribution distributes values such that approximately 68% of the values are within the standard deviation of the mean. The normal distribution models biometric data of a large population.

- The poisson distribution is a discrete distribution that models the number of occurrences, referred to as "arrivals", of an essentially infinite population of similar independent events per unit of distance, area, volume or time. You can use the poisson distribution to model the arrival of packets under steady-state conditions at a heavily-utilized router.

- The exponential distribution distributes values such that approximately 63% of the values are between 0 and the mean. Lower values are more likely than higher values. The exponential distribution models the interval between arrivals in a poisson process and can potentially model the size of packets arriving at a router, smaller packets being more frequent than larger packets.

Epsilon (E) is the *expected value*.

Mu (μ) is the *mean*.

Sigma (σ) is the *standard deviation*.

Lambda (γ) is the *arrival rate*

Testing Boundary Conditions

Test boundary conditions with worst-case tests that do the following:

- Test initial and terminal conditions.
- Focus on where the problems are.

Two Illustrations

```
reg [1:`WIDTH] memory [1:`SIZE];
...
for (i=1; i<=NUMBER; i=i+1)
 memory[i] = memory[i+1];
```

What is the architect's intent if NUMBER==0 ?

What is the architect's intent if NUMBER==`SIZE ?

```
always @(MEM_ACCESS)
 case ({READ, WRITE})
  1: memory[location] = ibus;
  2: obus = memory[location];
  3: ???
 endcase
```

What happens if READ and WRITE are both true?

cādence

The illustrations indicate what is meant by a worst-case condition. Every design has its own set of worst-case conditions: A queue becomes full or empty; an arbiter receives simultaneous requests. The set of worst-case conditions can be very large, and at the beginning of the project, not well understood.

Example Worst-Case Test – FIFO Model

```
module fifo ( output [1:8] data_o, output full, empty,
              input  [1:8] data_i, input load, unload, rst, clk );
 reg [1:8] busy, fifo[1:8];
 integer i;
 assign data_o = fifo[1], full = busy[8], empty = ~busy[1];
 always @(posedge clk)
  if (rst)
   busy = 0;
  else
   case ({load, unload})
    2'b01: for (i=1; i<=8; i=i+1) // - UNLOAD ONLY -
           begin
             fifo[i] <= (i==8)? 8'bx : fifo[i+1];
             busy[i] <= (i==8)? 1'b0 : busy[i+1];
           end
    2'b10: for (i=8; i>=1; i=i-1) // - LOAD ONLY -
           begin
             fifo[i] <= busy[i]? fifo[i] : data_i;
             busy[i] <= (i==1)? 1'b1 : busy[i-1];
           end
    2'b11: for (i=1; i<=8; i=i+1) // - LOAD AND UNLOAD -
             fifo[i] <= (i==8)? data_i : busy[i+1]? fifo[i+1] : data_i ;
   endcase
endmodule
```

1	2	3	4	5	6	7	8	b
2								u
3								s
4								y
5								b
6								i
7								t
8								s

cādence

The model treats the first and last locations differently than all other locations. Its treatment of the first and last locations is thus more susceptible to error, and the testbench must focus on testing model accuracy around those locations.

The model also becomes more complex in the area where it handles simultaneous read and write operations. This is an area that the model is more likely to inaccurate define, and the testbench is more likely to miss.

Example Worst-Case Test – FIFO Test Tasks

```verilog
module test ( output reg [1:8] data_i, output reg load, unload, rst,
              input wire [1:8] data_o, input wire full, empty, clk );

// - TASK TO CHECK FIFO DATA -
task checkit (input full_e, empty_e, input [1:8] data_e);
 if ( {full, empty} !== {full_e, empty_e}
  || !empty_e && data_o !== data_e ) begin
    $display("ERROR- FIFO TEST FAILED");
    $finish;
  end
endtask

// - TASK TO LOAD FIFO -
task loadit (input [1:8] data, input full_e, empty_e, input [1:8] data_e);
 begin
  @(negedge clk) load = 1; data_i = data;
  @(negedge clk) load = 0; data_i = 8'hz;
  checkit (full_e, empty_e, data_e);
 end
endtask

// - TASK TO UNLOAD FIFO -
task unloadit (input full_e, empty_e, input [1:8] data_e);
 begin
  @(negedge clk) unload = 1;
  @(negedge clk) unload = 0;
  checkit (full_e, empty_e, data_e);
 end
endtask
```

cādence

The testbench defines three tasks to off-load repetitive work from the main test block:

- The loadit task presents data to the FIFO and pulses the load signal.
- The unloadit task pulses the unload signal.
- The checkit task compares flags and data with expected values.

Example Worst-Case Test – FIFO Test Sequence

```verilog
task resetit;
 begin
  load = 0; unload = 0;
  @(negedge clk) rst = 1;
  @(negedge clk) rst = 0;
  checkit (0, 1, 8'hX);
 end
endtask

initial begin: TST
  integer i;
  resetit;                                  // Empty, not full
  loadit (1, 0, 0, 1);                      // Not empty, 1st data
  for (i=2; i<=7; i=i+1) loadit (i, 0, 0, 1); // Not yet full, still 1st data
  loadit ( 8, 1, 0, 1);                     // Now full, still 1st data
  loadit (-1, 1, 0, 1);                     // Overflow ignored
  unloadit (0, 0, 2);                       // No longer full, 2nd data
  for (i=3; i<=8; i=i+1) unloadit (0, 0, i); // Not yet empty, most data
  unloadit (0, 1, 8'hX);                    // Now empty
  unloadit (0, 1, 8'hX);                    // Underflow ignored
  $display ("FIFO TEST PASSED");
  $finish;
 end

endmodule
```

cadence

This sequence tests several FIFO boundary conditions. The test sequence verifies that the FIFO operates correctly, and also verifies that the FIFO does not operate incorrectly. The test sequence verifies:

- That the model sets the full flag only when the FIFO is full, and clears it only when the FIFO becomes not full.
- That the model sets the empty flag only when the FIFO is empty, and clears it only when the FIFO becomes not empty.
- That the model will not load data into a full FIFO, i.e., it does not change flags or data.
- That the model will not unload data from an empty FIFO, i.e., it does not change flags or data.

Testing Protocol Interactions

Test protocol interactions with timing offset ("sweep") tests, that:

- Vary the relative arrival of events to test all relevant combinations.
- Test the interaction between two or more independent functions.
- Serve some of the same purposes as random test sequences.
  - But efficiently test only a preset set of possible interactions!
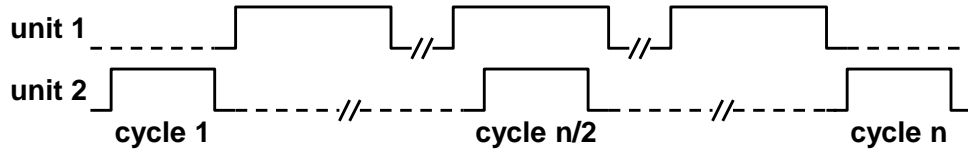
A sweep test sweeps one parameter across either a fixed point or another parameter, performing a regular search for combinations that the model may handle incorrectly.

A sweep test is ideal for test of a split-transaction bus, as it can verify that no phase of one transaction interferes with any phase of the other transaction.

Illustration of Sweep Test – Task Definitions

```verilog
module top;
 localparam PERIOD=20, LATENCY1=3;
 reg clk, request1, request2, busy1, busy2, done1, done2;

 always begin clk=1; #(PERIOD/2); clk=0; #(PERIOD/2); end

 task start_device1; // MIMIC A DEVICE WITH SOME LATENCY
  forever
   begin
    wait (request1) @(posedge clk) busy1 = 1;
    repeat (LATENCY1) @(posedge clk); done1 = 1; busy1 = 0;
    @(posedge clk) done1 = 0;
   end
 endtask

 task test_device1; // MAKE A REQUEST AND WAIT UNTIL DONE
  begin
   @(posedge clk) request1 = 1;
   @(posedge clk) request1 = 0;
   wait (done1) @(posedge clk);
  end
 endtask
```

Tasks for device 2 are similar so not shown.

The testbench defines two tasks to support the sweep test:

- The start_device task mimics a device with some latency. It continually waits for a request, outputs a busy flag, waits the defined latency time, de-asserts the busy flag and pulses the done flag.
- The test_device task pulses the request flag, waits for the done flag, and exits.

Illustration of Sweep Test – Test Sequence

```
initial
 fork
  start_device1;
  start_device2;
  begin: TEST
   integer cycles1, cycles2, i;
   cycles1 = $time/PERIOD;              // Compute device 1 latency
   test_device1;
   cycles1 = $time/PERIOD - cycles1;
   cycles2 = $time/PERIOD;              // Compute device 2 latency
   test_device2;
   cycles2 = $time/PERIOD - cycles2;
   for (i=cycles1; i<=cycles2+2*cycles1; i=i+1) // Sweep 2 across 1
    fork
     begin repeat(cycles1+cycles2) @(posedge clk); test_device1; end
     begin repeat(i               ) @(posedge clk); test_device2; end
    join
   $finish;
  end
 join

endmodule
```

unit 1

unit 2

cycle 1   cycle n/2   cycle n

cādence

The initial block forks simultaneous sequences to start both devices and to test them and their interaction with each other. The semi-intelligent test block determines the number of cycles needed by each device to complete its process, and then in a loop concurrently tests the devices. The loop starts device 1 at regular intervals and starts device 2 later for each iteration of the loop. The operation time of device 2 sweeps over the operation time of device 1, thereby testing any possible interference within this scenario.

Capturing and Playing Back Vectors

Why would you do vector capture and playback?

- To obtain and verify test vectors for off-line use, such as:
  - Fault detection analysis
  - ASIC foundry test machines

How would you do vector capture and playback?

- With system tasks and functions
  - Write textual data with a user-defined format
    - o **$display**, **$write**, **$monitor**, **$strobe**
  - Write textual data with a predefined format
    - o **$dump***
  - Read textual data with a predefined format
    - o **$readmem***, **$getpattern**

cādence

You can use any of the file output system tasks to save signal values. You can then message the output as desired, read the vectors into a memory, and play them back with your own Verilog code.

The value change dump (VCD) facility is more efficient than the $monitor task, in performance and in storage space. With the VCD facility, you can save value changes of variables for any portion of the design hierarchy during any specified time interval. You can save results globally, without having to explicitly name all signals. Simulators produce a textual dump file that contains header information, variable definitions, and the value changes for all variables specified in the task calls.

The IEEE Standard 1364-2001 describes but does not require the $getpattern system task. Not all simulators support this system task.

Vector Capture and Playback Issues

You must resolve these vector capture and playback issues:

- Timing:
  - For asynchronous circuits, capture and play back any value change.
  - For synchronous circuits, capture and play back the last value change during each clock cycle:
    - Determine the capture and playback time within the clock cycle.
    - Ensure signal stability when captured, what about negative setup/hold?
- Bidirectional signals:
  - Capture a control signal to determine playback direction.
    - Refer to timing considerations presented above.
- Vector width and depth:
  - Exercising internal functions may require many signal transitions.
  - Vector capture may require many large pattern files.

cādence

You must carefully and methodically develop a customized vector capture and playback mechanism. Think through, step by step, what you want to do and how to write Verilog to implement your plan.

You can use the VCD facility to save only the signal state changes, thus conserving storage space. If you have a fully synchronous subdesign, you can ignore all changes except the last one before the active clock edge on any non-clock signal, but you must ensure that the changes meet setup and hold requirements when played back.

You must capture control signals for all bidirectional signals so your playback mechanism can drive stimulus and check response appropriately.

Because test at the system level typically requires many clocks to do something useful, many vectors captured at the subdesign level may contain only clock changes. You can prune these vectors if you are either absolutely sure that they do not contribute to subdesign state changes, or you regenerate those clocks during pattern playback.

You should expect captured vector storage space to be an issue.

Capturing Vectors to a File

- You can capture the stimulus and response at the pins of your DUT.

- You can play these vectors back against the DUT using an efficient stripped-down test environment.

- You can provide these test vectors to the device manufacturer.

- You must resolve these issues:
  - Timing of the capture
  - Timing of the playback
  - Direction information

```verilog
module capture_tb;
 localparam PERIOD=10;
 reg  [7:0] stim_reg;
 wire [7:0] resp_net;
 DUT u1 (stim_reg, resp_net);
 initial begin: TEST
  // stimulus/response behaviors
  end
 initial begin: CAPTURE
  integer stimfile, respfile;
  stimfile = $fopen("stim.txt");
  respfile = $fopen("resp.txt");
  if (!stimfile || !respfile)
    // handle exception, else...
  forever #(PERIOD) begin
    $fstrobeb(stimfile,stim_reg);
    $fstrobeb(respfile,resp_net);
   end
  end
endmodule
```

cādence
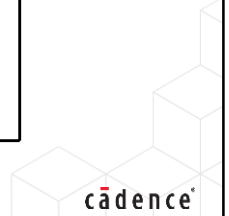
You can capture the stimulus and response at the pins of your DUT, and later play these vectors back against the DUT using an efficient stripped-down test environment. You can also provide these test vectors to the device manufacturer.

It is very important to time the capture to record the steady-state value of the pins.

You may need to also record design-dependent direction information for bidirectional pins.

This example opens stimulus and response files and simply records hopefully steady-state pin data on a periodic basis.

Applying Vectors from a File

You can preload stimulus values into an array and then in a loop apply the array values. Advantages include:

- Potential for run-time selection of tests from a test suite.

The file can contain:

- Manually or automatically generated test vectors.
- Captured test vectors.
- "Pseudocode" that your testbench executes to generate the stimulus.

```verilog
module playback_tb;
 localparam PERIOD=10;
 reg  [7:0] stim_reg;
 wire [7:0] resp_net;
 reg  [7:0] stim_mem[0:whatever];
 reg  [7:0] resp_mem[0:whatever];
 DUT u1 (stim_reg, resp_net);
 initial begin: TEST
  integer i;
  $readmemb("stim.txt",stim_mem);
  $readmemb("resp.txt",resp_mem);
  for (i=0;i<=whatever;i=i+1)
   begin
    stim_reg = stim_mem[i];
    #(PERIOD);
    if (resp_net!==resp_mem[i])
     // handle exception here ...
   end
  end
endmodule
```

cādence
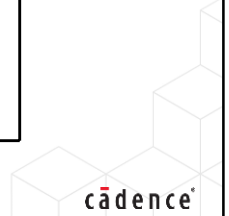
You can facilitate run-time selection of the test suite by applying stimulus from an array. You can place a forever loop in the testbench that loads and applies stimulus, and then waits for the user before continuing to the next loop iteration. The user can meanwhile change the file contents or use other implementation-specific features to replace the file.

As an alternative to directly applying file stimulus, you can incorporate a machine in your testbench to execute the pseudocode file to generate the stimulus.

This example loads stimulus vectors and expected response vectors from files, and then in a loop applies the stimulus and checks the response.

Example Vector Capture and Playback: Model

```verilog
module arbx ( inout [3:0] busa, busb,
              output gab, gba,
              input rab, rba, rst,
              input clk );
 localparam IDL=0, GAB=1, CLR=2, PRI=3, GBA=4;
 reg [2:0] state;
 assign busb=gab?busa:4'hz, busa=gba?busb:4'hz; // ASSIGN INOUT BUS
 assign gab=(state==GAB), gba=(state==GBA); // ASSIGN OUTPUT GRANT

 always @(posedge clk)
  if (rst)
   state <= IDL;
  else
   case (state)
    IDL: state <= rab?GAB:rba?GBA:IDL;
    GAB: state <= rab?GAB:CLR;
    CLR: state <= rab?GAB:rba?PRI:CLR;
    PRI: state <= rab?GAB:rba?GBA:IDL;
    GBA: state <= rba?GBA:CLR;
    default state <= IDL;
   endcase

endmodule
```
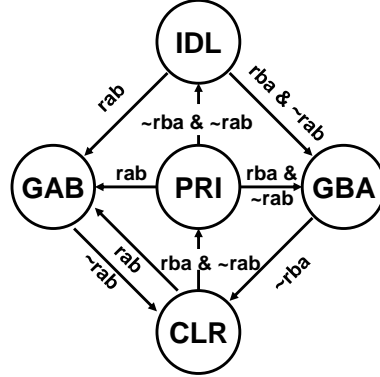
cādence®

This model is a combined arbiter and 4-bit bus transceiver. Direction A to B has absolute priority, and if direction B to A has just relinquished the transceiver and requests to retake it, it must wait one additional clock to give direction A to B the opportunity to use the bus first.

Example Vector Capture and Playback: Testbench

```verilog
`timescale 1 ns / 1 ns

module test
#( parameter PERIOD=10 )
 ( inout wire [3:0] busa, busb,
   input wire gab, gba,
   output reg rab, rba, rst,
   input wire clk );

 reg [3:0] busa_r, busb_r;
 assign busa = busa_r, busb = busb_r;

`ifdef CAPTURE
  `include "capture.inc"
`endif

`ifdef PLAYBACK
  `include "playback.inc"
`endif

endmodule
```

cadence

The testbench declares output variables for the busa and busb inout ports and connects them to the ports. The Verilog language does not permit you to directly declare bidirectional ports to be variables.

The testbench directly declares output variables for the rab and rba and rst output ports.

Example Vector Capture and Playback: Capture

```
initial begin: Capture
  integer mcd;
  time t;
  // PLAYBACK REPLICATES RESET SO DO NOT CAPTURE THIS
  rab=0; rba=0; busa_r=4'hz; busb_r=4'hz; rst=1;
  @(negedge clk) rst = 0;
  @(negedge clk);
  // START MONITOR AFTER RESET
  t = $time;
  mcd = $fopen ("patterns.inc");
  $timeformat (-9, 0, "", 3);
  $fmonitorb ( mcd, "#%t", ($time-t),
               "{rab,gab_r,rba,gba_r,busa_r,busb_r}<=12'b",
                 rab,gab   rba,gba,  busa,  busb, ";" );
  // ACTUAL TEST GOES HERE
  `include "stimulus.inc"
  @(negedge clk);
  // CLEAN UP AND GO HOME
  $monitoroff;
  $fdisplay ( mcd, "#%t $finish;", ($time-t) );
  $finish;
end
```

```
#  0{rab,gab_r,rba,gba_r,busa_r,busb_r}<=12'b0000zzzzzzzz;
                    ...
```

```
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h2zz; wait(gab);
                    ...
```

cādence

The capture code uses the $monitor system task to save a pattern vector at the end of each time slice in which any monitored signal changes. The capture code formats the $monitor system task to save signal changes in syntactically correct Verilog so that the simulator can easily play them back. This inefficient method to capture and play back signal changes is impractical for simulations of a significant size. You would probably want to save signal changes to a waveform database and use the Programming Language Interface (PLI) to read and reapply them as you go.

Example contents of stimulus.inc file:

```
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h2zz; wait(gab)
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h25z;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h2az;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h1zz; wait(gba)
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h1z3;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h1zc;
@(negedge clk) {rab, rba, busa_r, busb_r} = 10'h0zz; wait(!gba)
```

Module Summary

You should now be able to generate test stimulus.

This module discussed:

- Using hierarchical names
- Generating clocks
- Generating "in-line" stimulus
- Generating stimulus using loops
- Generating stimulus using tasks
- Generating random stimulus
- Testing "corner" conditions
- Testing protocol interactions
- Applying stimulus from a file

cādence®

This module examined some common coding styles and methods applicable to testbench generation.

Module Review

1. The Verilog-1995 standard permitted use of hierarchical names to reference any named objects. What Verilog-2001 constructs are incompatible with access by hierarchical names?

2. Even if you enable a task from only one place in your testbench, why might you *still* want to keep that set of statements in a task rather than "in-line" it at the point of the call?

3. Which random distribution is that infamous "bell curve" your teachers used to map numerical test scores into letter grades?

4. What are some of the issues to consider when capturing test vectors to a file and then playing them back?

cadence

This page does not contain notes.

Module Review Solutions

1. The Verilog-1995 standard permitted use of hierarchical names to reference any named objects. What Verilog-2001 constructs are incompatible with access by hierarchical names?

   - Hierarchical names cannot access items of automatic tasks or automatic functions because those items exist only while the task or function is active. Although unnamed generate blocks have a machine-generated implementation-dependent name for display purposes, source code is not permitted to know what that name is so cannot access items within unnamed generate blocks.

2. Even if you enable a task from only one place in your testbench, why might you *still* want to keep that set of statements in a task rather than "in-line" it at the point of the call?

   - For the same reason that you use an HDL instead of schematic capture, representing operations as abstract tasks makes it easier for you to understand the overall problem.

3. Which random distribution is that infamous "bell curve" your teachers used to map numerical test scores into letter grades?

   - The "bell curve" is a normal distribution.

4. What are some of the issues to consider when capturing test vectors to a file and then playing them back?

   - Issues to consider must include the stability of the captured signals and the setup and hold times when playing them back, how to record and apply the directionality of bidirectional signals, and perhaps the difference in clock speed.
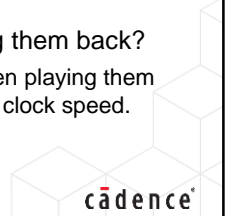
This page does not contain notes.

Lab

Lab 22-1   Verifying a Serial Interface Receiver

- For this lab, you will generate a simple test of the serial interface receiver.

- You will generate random data for four packets. You statically construct a 256-bit stream containing four valid packets that are surrounded by values that are something other than the header value.

- You will reset the receiver and present this stream to the receiver inputs. While adhering to the output protocol, you retrieve the receiver output data and verify that all packets are received and correct.

cādence

Your objective for this lab is to test a serial interface receiver.

The lab model is a serial interface receiver. The receiver interfaces between a serial input stream and a parallel output stream. A 24-bit packet represents the data. The packet consists of an 8-bit header with a value of 0xa5 followed by two 8-bit data bytes. The receiver initially shifts input data left into the header register until it detects the header value. For this reason, the header register must be initialized to a value opposite the leftmost header value bit. Upon detecting the packet header, the receiver clears the header register and shifts input data left into the body register while counting to 16. Upon the 16th count, the receiver moves the data to the output buffer, clears the counter, asserts the ready output, and again shifts input data left into the header register. The receiver can thus move a packet every 24 clocks.

The test environment reads the leftmost buffer byte while acknowledging the ready signal. The receiver shifts the output buffer left by one byte upon each such acknowledge. The receiver counts acknowledges and drops the ready signal upon the last acknowledge. The test environment can delay the last acknowledge up to and including the clock that loads the next data into the output buffer. Failure of the test environment to retrieve data within that interval results in lost data.

For this lab, you generate a simple test of the serial interface receiver. You generate random data for four packets. You statically construct a 256-bit stream containing four valid packets that are surrounded by values that are something other than the header value. You reset the receiver and present this stream to the receiver inputs. While adhering to the output protocol, you retrieve the receiver output data and verify that all packets are received and correct.