Chapter 11
A Complete SystemVerilog Testbench

This chapter applies the many concepts you have learned about SystemVerilog features to verify a design. The testbench creates constrained random stimulus, and gathers functional coverage. It is structured according to the guidelines from Chapter 8 so you can inject new behavior without modifying the lower-level blocks.

The design is an ATM switch that was shown in Sutherland [2006], who based his SystemVerilog description on an example from Janick Bergeron's Verification Guild. Sutherland took the original Verilog design and used SystemVerilog design features to create a switch that can be configured from 4×4 to 16×16. The testbench in the original example creates ATM cells using `$urandom`, overwrites certain fields with ID values, sends them through the device, then checks that the same values were received.

The entire example, with the testbench and ATM switch, is available for download at `http://chris.spear.net/systemverilog`. This chapter shows just the testbench code.

11.1 Design Blocks

The overall connection between the design and testbench, shown in Fig. 11.1, follows the pattern shown in Chapter 4.

C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, DOI 10.1007/978-1-4614-0715-7_11,
© Springer Science+Business Media, LLC 2012

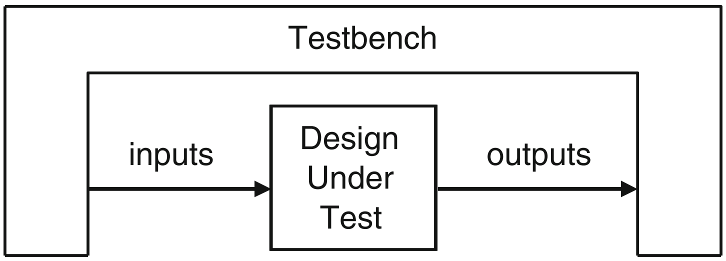


Fig. 11.1 The testbench — design environment

The top level of the design is called `squat`, as shown in Fig. 11.2. The module has 1..N Utopia Rx interfaces that are sending UNI formatted cells. Inside the DUT, cells are stored, converted to NNI format, and forwarded to the Tx interfaces. The forwarding is done according to a lookup table that is addressed with the VPI field of the incoming cell. The table is programmed through the management interface.

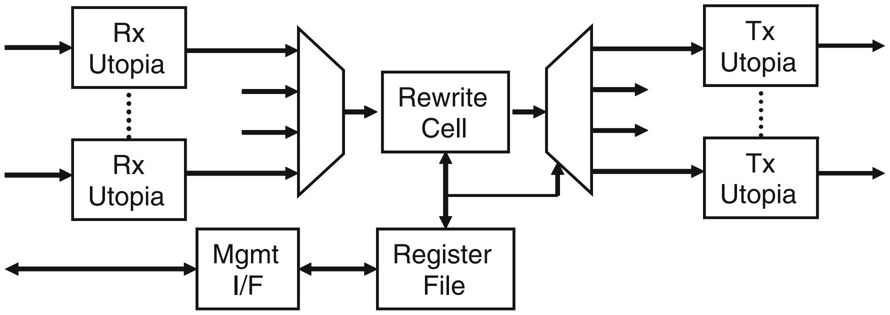


Fig. 11.2 Block diagram for the squat design

The top level module in Sample 11.1 defines arrays of interfaces for the Rx and Tx ports.

Sample 11.1 Top level module

```
`timescale 1ns/1ns
`define TxPorts 4  // set number of transmit ports
`define RxPorts 4  // set number of receive ports

module top;
  parameter int NumRx = `RxPorts;
  parameter int NumTx = `TxPorts;

  logic rst, clk;
  // System Clock and Reset
  initial begin
    rst = 0; clk = 0;
    #5ns rst = 1;
    #5ns clk = 1;
    #5ns rst = 0; clk = 0;
    forever
      #5ns clk = ~clk;
  end

  Utopia Rx[0:NumRx-1] ();// NumRx x Level 1 Utopia Rx Interface
  Utopia Tx[0:NumTx-1] ();// NumTx x Level 1 Utopia Tx Interface
  cpu_ifc mif();          // Utopia management interface
  squat #(NumRx, NumTx) squat(Rx, Tx, mif, rst, clk);  // DUT
  test  #(NumRx, NumTx) t1(Rx, Tx, mif, rst);          // Test
endmodule : top
```

The testbench program in Sample 11.2 passes the interfaces and signals through the port list. See Section 10.1.4 for a discussion on ports vs. cross module references. The actual testbench code is in the `Environment` class. The program steps through the phases of the environment. In order to work at a higher level of abstraction, the testbench only uses clocking blocks in the interfaces to synchronize with the DUT, not low level clocks.

Sample 11.2 Testbench program

```
program automatic test
  #(parameter int NumRx = 4, parameter int NumTx = 4)
   (Utopia.TB_Rx Rx[0:NumRx-1],
    Utopia.TB_Tx Tx[0:NumTx-1],
    cpu_ifc.Test mif,
    input logic rst);

`include "environment.sv"
  Environment env;

  initial begin
    env = new(Rx, Tx, NumRx, NumTx, mif);
    env.gen_cfg();
    env.build();
    env.run();
    env.wrap_up();
  end
endprogram // test
```

The testbench loads control information into the ATM switch through the Management interface, also known as the CPU interface, shown in Sample 11.3. In this chapter's examples, the interface is only used to load the lookup table that maps VPI to forwarding masks.

Sample 11.3 CPU Management Interface

```
interface cpu_ifc;
  logic        BusMode, Sel, Rd_DS, Wr_RW, Rdy_Dtack;
  logic [11:0] Addr;
  CellCfgType  DataIn, DataOut;    // Defined in Sample 11-11

  modport Peripheral
          (input  BusMode, Addr, Sel, DataIn, Rd_DS, Wr_RW,
           output DataOut, Rdy_Dtack);

  modport Test
          (output BusMode, Addr, Sel, DataIn, Rd_DS, Wr_RW,
           input  DataOut, Rdy_Dtack);

endinterface : cpu_ifc

typedef virtual cpu_ifc.Test vCPU_T;
```

Sample 11.4 shows the Utopia interface, which is used by the testbench to communicate with the squat design by transmitting and receiving ATM cells. The interface has clocking blocks for the transmit and receive paths, and modports for the design and testbench connections to the interface.

Sample 11.4 Utopia interface

```
interface Utopia #(IfWidth = 8);

  logic [IfWidth-1:0] data;
  bit clk_in, clk_out;
  bit soc, en, clav, valid, ready, reset, selected;

  ATMCellType ATMcell;  // union of structures for ATM cells

  modport TopReceive (
    input  data, soc, clav,
    output clk_in, reset, ready, clk_out, en, ATMcell, valid );

  modport TopTransmit (
    input  clav,
    inout  selected,
    output clk_in, clk_out, ATMcell, data, soc, en, valid,
           reset, ready );

  modport CoreReceive (
    input  clk_in, data, soc, clav, ready, reset,
    output clk_out, en, ATMcell, valid );

  modport CoreTransmit (
    input  clk_in, clav, ATMcell, valid, reset,
    output clk_out, data, soc, en, ready );

   clocking cbr @(negedge clk_out);
      input clk_in, clk_out, ATMcell, valid, reset, en, ready;
      output data, soc, clav;
   endclocking : cbr
   modport TB_Rx (clocking cbr);

   clocking cbt @(negedge clk_out);
      input  clk_out, clk_in, ATMcell, soc, en, valid,
             reset, data, ready;
      output clav;
   endclocking : cbt
   modport TB_Tx (clocking cbt);

endinterface

typedef virtual Utopia vUtopia;
typedef virtual Utopia.TB_Rx vUtopiaRx;
typedef virtual Utopia.TB_Tx vUtopiaTx;
```

11.2 Testbench Blocks

The environment class, as shown in Section 8.2.1, is the scaffolding that supports the testbench structure. Inside this class lies the blocks of your layered testbench, such as generators, drivers, monitors, and scoreboard. The environment also controls the sequencing of the four testbench steps: generate a random configuration, build the testbench environment, run the test and wait for it to complete, and a wrap-up phase to shut down the system and generate reports. Sample 11.5 shows the ATM environment class. It uses the virtual interface `vCPU_T` defined in Sample 11.3.

Sample 11.5 Environment class header

```
class Environment;
  UNI_generator gen[];
  mailbox gen2drv[];
  event   drv2gen[];
  Driver drv[];
  Monitor mon[];
  Config cfg;
  Scoreboard scb;
  Coverage cov;
  virtual Utopia.TB_Rx Rx[];
  virtual Utopia.TB_Tx Tx[];
  int numRx, numTx;
  vCPU_T mif;
  CPU_driver cpu;

  extern function new(input vUtopiaRx Rx[],
                      input vUtopiaTx Tx[],
                      input int numRx, numTx,
                      input vCPU_T mif);
  extern virtual function void gen_cfg();
  extern virtual function void build();
  extern virtual task run();
  extern virtual function void wrap_up();

endclass : Environment
```

With the `$test$plusargs()` system task, the `Environment` class constructor in Sample 11.6 looks for the VCS switch `+ntb_random_seed`, which sets the random seed for the simulation. The system task `$value$plusargs()` extracts the value from the switch. Your simulator may have a different way to set the seed. It is important to print the seed in the log file so that if the test fails, you can run it again with the same value.

Sample 11.6 Environment class methods

```
//---------------------------------------------------------------
// Construct an environment instance
function Environment::new(input vUtopiaRx Rx[],
                          input vUtopiaTx Tx[],
                          input int numRx, numTx,
                          input vCPU_T mif);
  this.Rx = new[Rx.size()];
  foreach (Rx[i]) this.Rx[i] = Rx[i];
  this.Tx = new[Tx.size()];
  foreach (Tx[i]) this.Tx[i] = Tx[i];
  this.numRx = numRx;
  this.numTx = numTx;
  this.mif = mif;
  cfg = new(numRx,numTx);

  if ($test$plusargs("ntb_random_seed")) begin
    int seed;
    $value$plusargs("ntb_random_seed=%d", seed);
    $display("Simulation run with random seed=%0d", seed);
  end
  else
    $display("Simulation run with default random seed");
endfunction : new

//---------------------------------------------------------------
// Randomize the configuration descriptor
function void Environment::gen_cfg();
  `SV_RAND_CHECK(cfg.randomize());
  cfg.display();
endfunction : gen_cfg

//---------------------------------------------------------------
// Build the environment objects for this test
// Note that objects are built for every channel,
// even if they are not used.  This reduces null handle bugs.
function void Environment::build();
  cpu = new(mif, cfg);
  gen = new[numRx];
  drv = new[numRx];
  gen2drv = new[numRx];
  drv2gen = new[numRx];
  scb = new(cfg);
  cov = new();

  // Build generators
  foreach(gen[i]) begin
    gen2drv[i] = new();
    gen[i] = new(gen2drv[i], drv2gen[i],
                 cfg.cells_per_chan[i], i);
    drv[i] = new(gen2drv[i], drv2gen[i], Rx[i], i);
  end

  // Build monitors
  mon = new[numTx];
  foreach (mon[i])
    mon[i] = new(Tx[i], i);

  // Connect scoreboard to drivers & monitors with callbacks
  begin
    Scb_Driver_cbs sdc = new(scb);
    Scb_Monitor_cbs smc = new(scb);
    foreach (drv[i]) drv[i].cbsq.push_back(sdc);
    foreach (mon[i]) mon[i].cbsq.push_back(smc);
  end

  // Connect coverage to monitor with callbacks
  begin
    Cov_Monitor_cbs smc = new(cov);
   foreach (mon[i])
     mon[i].cbsq.push_back(smc);
  end
endfunction : build

//---------------------------------------------------------------
// Start the transactors: generators, drivers, monitors
// Channels that are not in use don't get started
task Environment::run();
  int num_gen_running;

  // The CPU interface initializes before anyone else
  cpu.run();

  num_gen_running = numRx;

  // For each input RX channel, start generator and driver
  foreach(gen[i]) begin
    int j=i;     // Automatic var holds index in spawned threads
    fork
      begin
        if (cfg.in_use_Rx[j])
          gen[j].run();          // Wait for generator to finish
        num_gen_running--;// Decrement driver count
     end
     if (cfg.in_use_Rx[j]) drv[j].run();
    join_none
  end
  // For each output TX channel, start monitor
  foreach(mon[i]) begin
    int j=i;     // Automatic var holds index in spawned threads
    fork
      mon[j].run();
    join_none
  end

  // Wait for all generators to finish, or time-out
  fork : timeout_block
    wait (num_gen_running == 0);
    begin
      repeat (1_000_000) @(Rx[0].cbr);
      $display("@%0t: %m ERROR: Generator timeout ", $time);
      cfg.nErrors++;
    end
  join_any
  disable timeout_block;

  // Wait for the data to flow through switch, into monitors,
  // and scoreboards
  repeat (1_000) @(Rx[0].cbr);
endtask : run

//---------------------------------------------------------------
// Post-run cleanup / reporting
function void Environment::wrap_up();
  $display("@%0t: End of sim, %0d errors, %0d warnings",
           $time, cfg.nErrors, cfg.nWarnings);
  scb.wrap_up();
endfunction : wrap_up
```

The method `Environment::build` in Sample 11.6 connects the scoreboard to the driver and monitor with the callback class, which is shown in Sample 11.7, `Scb_Driver_cbs`. This class sends the expected values to the scoreboard. The base driver callback class, `Driver_cbs`, is shown in Sample 11.20.

Sample 11.7 Callback class connects driver and scoreboard

```
class Scb_Driver_cbs extends Driver_cbs;
  Scoreboard scb;

  function new(input Scoreboard scb);
    this.scb = scb;
  endfunction : new

  // Send received cell to scoreboard
  virtual task post_tx(input Driver drv,
    input UNI_cell c);
    scb.save_expected(c);
  endtask : post_tx
endclass : Scb_Driver_cbs
```

The callback class in Sample 11.8, `Scb_Monitor_cbs`, connects the monitor with the scoreboard. The base monitor callback class, `Monitor_cbs`, is shown in Sample 11.21.

Sample 11.8 Callback class connects monitor and scoreboard

```
class Scb_Monitor_cbs extends Monitor_cbs;
  Scoreboard scb;

  function new(input Scoreboard scb);
    this.scb = scb;
  endfunction : new

  // Send received cell to scoreboard
  virtual task post_rx(input Monitor mon,
                       input NNI_cell c);
    scb.check_actual(c, mon.PortID);
  endtask : post_rx
endclass : Scb_Monitor_cbs
```

The environment connects the monitor to the coverage class with the final callback class, `Cov_Monitor_cbs`, shown in Sample 11.9.

Sample 11.9 Callback class connects the monitor and coverage

```
class Cov_Monitor_cbs extends Monitor_cbs;
  Coverage cov;

  function new(input Coverage cov);
    this.cov = cov;
  endfunction : new

  // Send received cell to coverage
  virtual task post_rx(input Monitor mon,
                       input NNI_cell c);
    CellCfgType CellCfg = top.squat.lut.read(c.VPI);
    cov.sample(mon.PortID, CellCfg.FWD);
  endtask : post_rx
endclass : Cov_Monitor_cbs
```

The random configuration class header is shown in Sample 11.10. It starts with `nCells`, a random value for the total number of cells that flow through the system. The constraint `c_nCells_valid` ensures that the number of cells is valid by being greater than zero, whereas `c_nCells_reasonable` limits the test to a reasonable size, 1000 cells. You can disable or override this if you want longer tests.

Next is a dynamic bit array, `in_use_Rx`, to specify which Rx channels into the switch are active. This is used in Sample 11.6 in the `run` method so that only active channels run.

The array `cells_per_chan` is used to randomly divide the total number of cells across the active channels. The constraint `zero_unused_channels` sets the number of cells to zero for inactive channels. To help the solver, the active channel mask is solved before dividing up the cells between channels. Otherwise, a channel would be inactive only if the number of cells assigned to it was zero, which is very unlikely.

Sample 11.10 Environment configuration class

```
class Config;
  int nErrors, nWarnings;  // Number of errors, warnings
  bit [31:0] numRx, numTx; // Copy of parameters

  rand bit [31:0] nCells;  // Total cells
  constraint c_nCells_valid
     {nCells > 0; }
  constraint c_nCells_reasonable
     {nCells < 1000; }

  rand bit in_use_Rx[];     // Input / output channel enabled
  constraint c_in_use_valid
    {in_use_Rx.sum() > 0; }   // At least one RX is enabled

  rand bit [31:0] cells_per_chan[];
  constraint c_sum_ncells_sum  // Split cells over all channels
    {cells_per_chan.sum() == nCells;}   // Total number of cells

  // Set the cell count to zero for any channel not in use
  constraint zero_unused_channels
    {foreach (cells_per_chan[i])
      {
       // Needed for even dist of in_use
       solve in_use_Rx[i] before cells_per_chan[i];
       if (in_use_Rx[i])
         cells_per_chan[i] inside {[1:nCells]};
       else cells_per_chan[i] == 0;
      }
    }

  extern function new(input bit [31:0] numRx, numTx);
  extern virtual function void display(input string prefix="");
endclass : Config
```

The cell rewriting and forwarding configuration type is shown in Sample 11.11.

Sample 11.11 Cell configuration type

```
typedef struct packed {
  bit [`TxPorts-1:0] FWD;
  bit [11:0] VPI;
} CellCfgType;
```

The methods for the configuration class are shown in Sample 11.12

Sample 11.12 Configuration class methods

```
function Config::new(input bit [31:0] numRx, numTx);
  this.numRx = numRx;
  in_use_Rx = new[numRx];
  this.numTx = numTx;
  cells_per_chan = new[numRx];
endfunction : new

function void Config::display(input string prefix);
  $write("%sConfig: numRx=%0d, numTx=%0d, nCells=%0d (",
          prefix, numRx, numTx, nCells);
  foreach (cells_per_chan[i])
    $write("%0d ", cells_per_chan[i]);
  $write("), enabled RX: ", prefix);
  foreach (in_use_Rx[i]) if (in_use_Rx[i]) $write("%0d ", i);
  $display;
endfunction : display
```

The ATM switch accepts UNI formatted cells and sends out NNI formatted cells. These cells are sent through both an OOP testbench and a structural design, so they are defined using `typedef`. The major difference between the two formats is that the UNI's GFC and VPI field are merged into the NNI's VPI. The definitions in Sample 11.13 through 11.15 are from Sutherland [2006].

Sample 11.13 UNI cell format

```
typedef struct packed {
  bit         [3:0]  GFC;
  bit         [7:0]  VPI;
  bit         [15:0] VCI;
  bit                CLP;
  bit         [2:0]  PT;
  bit         [7:0]  HEC;
  bit [0:47]  [7:0]  Payload;
} uniType;
```

Sample 11.14 NNI cell format

```
typedef struct packed {
  bit         [11:0] VPI;
  bit         [15:0] VCI;
  bit                CLP;
  bit         [2:0]  PT;
  bit         [7:0]  HEC;
  bit [0:47]  [7:0]  Payload;
} nniType;
```

The UNI and NNI cells are merged with a byte memory to form a universal type, shown in Sample 11.15.

Sample 11.15 ATMCellType

```
typedef union packed {
  uniType uni;
  nniType nni;
  bit [0:52] [7:0] Mem;
} ATMCellType;
```

The testbench generates constrained random ATM cells, shown in Sample 11.16, that are extended from the `BaseTr` class, defined in Sample 8.24.

Sample 11.16 UNI_cell definition

```
class UNI_cell extends BaseTr;
  // Physical fields
  rand bit         [3:0]  GFC;
  rand bit         [7:0]  VPI;
  rand bit         [15:0] VCI;
  rand bit                CLP;
  rand bit         [2:0]  PT;
       bit         [7:0]  HEC;
  rand bit [0:47]  [7:0]  Payload;

  // Meta-data fields
  static bit [7:0] syndrome[0:255];
  static bit syndrome_not_generated = 1;

  extern function new();
  extern function void post_randomize();
  extern virtual function bit compare(input BaseTr to);
  extern virtual function void display(input string prefix="");
  extern virtual function BaseTr copy(input BaseTr to=null);
  extern virtual function void pack(output ATMCellType to);
  extern virtual function void unpack(input ATMCellType from);
  extern function NNI_cell to_NNI();
  extern function void generate_syndrome();
  extern function bit [7:0] hec (bit [31:0] hdr);
endclass : UNI_cell
```

Sample 11.17 shows the methods for the UNI cell.

Sample 11.17 UNI_cell methods

```
function UNI_cell::new();
  if (syndrome_not_generated)
    generate_syndrome();
endfunction : new


// Compute the HEC value after all other data has been chosen
function void UNI_cell::post_randomize();
  HEC = hec({GFC, VPI, VCI, CLP, PT});
endfunction : post_randomize


// Compare this cell with another
// This could be improved by telling what field mismatched
function bit UNI_cell::compare(input BaseTr to);
  UNI_cell c;
  $cast(c, to);
  if (this.GFC != c.GFC)         return 0;
  if (this.VPI != c.VPI)         return 0;
  if (this.VCI != c.VCI)         return 0;
  if (this.CLP != c.CLP)         return 0;
  if (this.PT  != c.PT)          return 0;
  if (this.HEC != c.HEC)         return 0;
  if (this.Payload != c.Payload) return 0;
  return 1;
endfunction : compare


// Print a "pretty" version of this object
function void UNI_cell::display(input string prefix);
  ATMCellType p;

  $display("%sUNI id:%0d GFC=%x, VPI=%x, VCI=%x, CLP=%b, PT=%x,
HEC=%x, Payload[0]=%x",
           prefix, id, GFC, VPI, VCI, CLP, PT, HEC, Payload[0]);
  this.pack(p);
  $write("%s", prefix);
  foreach (p.Mem[i]) $write("%x ", p.Mem[i]);
  $display;
endfunction : display


// Make a copy of this object
function BaseTr UNI_cell::copy(input BaseTr to);
  if (to == null) copy = new();
  else            $cast(copy, to);
  copy.GFC     = this.GFC;
  copy.VPI     = this.VPI;
  copy.VCI     = this.VCI;
  copy.CLP     = this.CLP;
  copy.PT      = this.PT;
  copy.HEC     = this.HEC;
  return copy;
endfunction : copy


// Pack this object's properties into a byte array
function void UNI_cell::pack(output ATMCellType to);
  to.uni.GFC     = this.GFC;
  to.uni.VPI     = this.VPI;
  to.uni.VCI     = this.VCI;
  to.uni.CLP     = this.CLP;
  to.uni.PT      = this.PT;
  to.uni.HEC     = this.HEC;
  to.uni.Payload = this.Payload;
endfunction : pack


// Unpack a byte array into this object
function void UNI_cell::unpack(input ATMCellType from);
  this.GFC     = from.uni.GFC;
  this.VPI     = from.uni.VPI;
  this.VCI     = from.uni.VCI;
  this.CLP     = from.uni.CLP;
  this.PT      = from.uni.PT;
  this.HEC     = from.uni.HEC;
  this.Payload = from.uni.Payload;
endfunction : unpack


// Generate a NNI cell from an UNI cell - used in scoreboard
function NNI_cell UNI_cell::to_NNI();
  NNI_cell copy;
  copy = new();
  copy.VPI     = this.VPI;   // NNI has wider VPI
  copy.VCI     = this.VCI;
  copy.CLP     = this.CLP;
  copy.PT      = this.PT;
  copy.HEC     = this.HEC;
  copy.Payload = this.Payload;
  return copy;
endfunction : to_NNI
```

```
// Generate the syndrome array, which is used to compute HEC
function void UNI_cell::generate_syndrome();
   bit [7:0] sndrm;
   for (int i = 0; i < 256; i = i + 1 ) begin
      sndrm = i;
      repeat (8) begin
         if (sndrm[7] === 1'b1)
           sndrm = (sndrm << 1) ^ 8'h07;
         else
           sndrm = sndrm << 1;
      end
      syndrome[i] = sndrm;
   end
   syndrome_not_generated = 0;
endfunction : generate_syndrome


// Compute the HEC value for this object
function bit [7:0] UNI_cell::hec (input bit [31:0] hdr);
   hec = 8'h00;
   repeat (4) begin
      hec = syndrome[hec ^ hdr[31:24]];
      hdr = hdr << 8;
   end
   hec = hec ^ 8'h55;
endfunction : hec
```

The `NNI_cell` class is almost identical to `UNI_cell`, except that it does not have a GFC field, or a method to convert to a `UNI_cell`.

Sample 11.18 shows the UNI cells random atomic generator, as originally shown in Section 8.2. The generator randomizes the blueprint instance of the UNI cell, and then sends out a copy of the cell to the driver.

Sample 11.18 UNI_generator class

```
class UNI_generator;
  UNI_cell blueprint; // Blueprint for generator
  mailbox  gen2drv;   // Mailbox to driver for cells
  event    drv2gen;   // Event from driver when done with cell
  int      nCells;    // Num cells for this generator to create
  int      PortID;    // Which Rx port are we generating?

  function new(input mailbox gen2drv,
               input event drv2gen,
               input int nCells, PortID);
    this.gen2drv = gen2drv;
    this.drv2gen = drv2gen;
    this.nCells  = nCells;
    this.PortID  = PortID;
    blueprint = new();
  endfunction : new

  task run();
    UNI_cell c;
    repeat (nCells) begin
      `SV_RAND_CHECK(blueprint.randomize());
      $cast(c, blueprint.copy());
      c.display($sformatf("@%0t: Gen%0d: ", $time, PortID));
      gen2drv.put(c);
      @drv2gen;// Wait for driver to finish with it
    end
  endtask : run

endclass : UNI_generator
```

Sample 11.19 shows the Driver class that sends UNI cells into the ATM switch. This class uses the driver callbacks in Sample 11.20. Note that there is a circular relationship here. The `Driver` class has a queue of `Driver_cbs` objects, and the `pre_tx()` and `post_tx()` methods in `Driver_cbs` are passed `Driver` objects. When you compile the two classes, you may need either `typedef class Driver;` before the `Driver_cbs` class definition, or `typedef class Driver_cbs;` before the `Driver` class definition.

Sample 11.19 driver class

```
typedef class Driver_cbs;

class Driver;

  mailbox gen2drv;     // For cells sent from generator
  event   drv2gen;     // Tell generator when I am done with cell
  vUtopiaRx Rx;        // Virtual ifc for transmitting cells
  Driver_cbs cbsq[$];  // Queue of callback objects
  int PortID;

  extern function new(input mailbox gen2drv,
                      input event drv2gen,
                      input vUtopiaRx Rx,
                      input int PortID);
  extern task run();
  extern task send (input UNI_cell c);

endclass : Driver


// new(): Construct a driver object
function Driver::new(input mailbox gen2drv,
                     input event drv2gen,
                     input vUtopiaRx Rx,
                     input int PortID);
  this.gen2drv = gen2drv;
  this.drv2gen = drv2gen;
  this.Rx      = Rx;
  this.PortID  = PortID;
endfunction : new


// run(): Run the driver.
// Get transaction from generator, send into DUT
task Driver::run();
  UNI_cell c;
  bit drop = 0;

  // Initialize ports
  Rx.cbr.data  <= 0;
  Rx.cbr.soc   <= 0;
  Rx.cbr.clav  <= 0;

  forever begin
    // Read the cell at the front of the mailbox
    gen2drv.peek(c);
    begin: Tx
      // Pre-transmit callbacks
      foreach (cbsq[i]) begin
        cbsq[i].pre_tx(this, c, drop);
        if (drop) disable Tx; // Don't transmit this cell
      end

    c.display($sformatf("@%0t: Drv%0d: ", $time, PortID));
    send(c);
    // Post-transmit callbacks
    foreach (cbsq[i])
      cbsq[i].post_tx(this, c);
    end : Tx

    gen2drv.get(c);  // Remove cell from the mailbox
    ->drv2gen;  // Tell the generator we are done with this cell
   end
endtask : run


// send(): Send a cell into the DUT
task Driver::send(input UNI_cell c);
  ATMCellType Pkt;

  c.pack(Pkt);
  $write("Sending cell: ");
  foreach (Pkt.Mem[i])
  $write("%x ", Pkt.Mem[i]); $display;

  // Iterate thru bytes of cell
  @(Rx.cbr);
  Rx.cbr.clav <= 1;
  for (int i=0; i<=52; i++) begin
    // If not enabled, loop
    while (Rx.cbr.en === 1'b1) @(Rx.cbr);

    // Assert Start Of Cell, assert enable, send byte 0 (i==0)
    Rx.cbr.soc  <= (i == 0);
    Rx.cbr.data <= Pkt.Mem[i];
    @(Rx.cbr);
  end
  Rx.cbr.soc <= 'z;
  Rx.cbr.data <= 8'bx;
  Rx.cbr.clav <= 0;
endtask
```

Sample 11.20 shows the driver callback class which has simple callbacks that are called before & after a cell is transmitted. This class has empty tasks, which are used by default. A test case can extend this class to inject new behavior in the driver without having to change any code in the driver

Sample 11.20 Driver callback class

```
typedef class Driver;

class Driver_cbs;
  virtual task pre_tx(input Driver drv,
                      input UNI_cell c,
                      inout bit drop);
  endtask : pre_tx

  virtual task post_tx(input Driver drv,
                       input UNI_cell c);
  endtask : post_tx
endclass : Driver_cbs
```

The `Monitor` class in Sample 11.21 has a very simple callback, with just one task that is called after a cell is received.

Sample 11.21 Monitor callback class

```
typedef class Monitor;

class Monitor_cbs;
  virtual task post_rx(input Monitor mon,
                       input NNI_cell c);
  endtask : post_rx
endclass : Monitor_cbs
```

Sample 11.22 shows the `Monitor` class. Like the `Driver` class, this uses a `typedef` to break the circular compile dependency with `Monitor_cbs`.

Sample 11.22 The Monitor class

```
typedef class Monitor_cbs;

class Monitor;

  vUtopiaTx Tx;           // Virtual interface with output of DUT
  Monitor_cbs cbsq[$];    // Queue of callback objects
  bit [1:0] PortID;

  extern function new(input vUtopiaTx Tx, input int PortID);
  extern task run();
  extern task receive (output NNI_cell c);
endclass : Monitor


// new(): construct an object
function Monitor::new(input vUtopiaTx Tx, input int PortID);
  this.Tx     = Tx;
  this.PortID = PortID;
endfunction : new


// run(): Run the monitor
task Monitor::run();
  NNI_cell c;

  forever begin
    receive(c);
    foreach (cbsq[i])
      cbsq[i].post_rx(this, c);  // Post-receive callback
  end
endtask : run


// receive(): Read cell from the DUT, pack into a NNI cell
task Monitor::receive(output NNI_cell cell);
  ATMCellType Pkt;

  Tx.cbt.clav <= 1;
  while (Tx.cbt.soc !== 1'b1 && Tx.cbt.en !== 1'b0)
    @(Tx.cbt);
  for (int i=0; i<=52; i++) begin
    // If not enabled, loop
    while (Tx.cbt.en !== 1'b0) @(Tx.cbt);

    Pkt.Mem[i] = Tx.cbt.data;
    @(Tx.cbt);
  end

  Tx.cbt.clav <= 0;

  c = new();
  c.unpack(Pkt);
  c.display($sformatf("@%0t: Mon%0d: ", $time, PortID));
endtask : receive
```

The scoreboard in Sample 11.23 gets expected cells from the driver through the function `save_expected`, and the cells actually received by the monitor with the function `check_actual`. The function `save_expected()` is called from the callback `Scb_Driver_cbs::post_tx()`, shown in Sample 11.7. The function `check_actual()` is called from `Scb_Monitor_cbs::post_rx()` in Sample 11.8.

Sample 11.23 The Scoreboard class

```
class Expect_cells;
  NNI_cell q[$];
  int iexpect, iactual;
endclass : Expect_cells


class Scoreboard;
  Config cfg;
  Expect_cells expect_cells[];
  NNI_cell cellq[$];
  int iexpect, iactual;

  extern function new(Config cfg);
  extern virtual function void wrap_up();
  extern function void save_expected(UNI_cell ucell);
  extern function void check_actual(input NNI_cell c,
                                    input int portn);
  extern function void display(string prefix="");
endclass : Scoreboard


function Scoreboard::new(input Config cfg);
  this.cfg = cfg;
  expect_cells = new[NumTx];
  foreach (expect_cells[i])
    expect_cells[i] = new();
endfunction : Scoreboard


function void Scoreboard::save_expected(input UNI_cell ucell);
  NNI_cell ncell = ucell.to_NNI;
  CellCfgType CellCfg = top.squat.lut.read(ncell.VPI);

  $display("@%0t: Scb save: VPI=%0x, Forward=%b",
           $time, ncell.VPI, CellCfg.FWD);
  ncell.display($sformatf("@%0t: Scb save: ", $time));

  // Find all Tx ports where this cell will be forwarded
  for (int i=0; i<NumTx; i++)
    if (CellCfg.FWD[i]) begin
      expect_cells[i].q.push_back(ncell); // Save cell in this q
      expect_cells[i].iexpect++;
      iexpect++;
    end
endfunction : save_expected
```

```
function void Scoreboard::check_actual(input NNI_cell c,
                                       input int portn);
  NNI_cell match;
  int match_idx;

  c.display($sformatf("@%0t: Scb check: ", $time));

  if (expect_cells[portn].q.size() == 0) begin
    $display("@%0t: ERROR: %m cell not found, SCB TX%0d empty",
             $time, portn);
    c.display("Not Found: ");
    cfg.nErrors++;
    return;
  end

  expect_cells[portn].iactual++;
  iactual++;

  foreach (expect_cells[portn].q[i]) begin
    if (expect_cells[portn].q[i].compare(c)) begin
      $display("@%0t: Match found for cell", $time);
      expect_cells[portn].q.delete(i);
      return;
    end
  end

  $display("@%0t: ERROR: %m cell not found", $time);
  c.display("Not Found: ");
  cfg.nErrors++;
endfunction : check_actual


// Print end of simulation report
function void Scoreboard::wrap_up();
  $display("@%0t: %m %0d expected cells, %0d actual cells rcvd",
           $time, iexpect, iactual);

  // Look for leftover cells
  foreach (expect_cells[i]) begin
    if (expect_cells[i].q.size()) begin
      $display("@%0t: %m cells in SCB Tx[%0d] at end of test",
               $time, i);
      this.display("Unclaimed: ");
      cfg.nErrors++;
    end
  end
endfunction : wrap_up
```

```
// Print the contents of the scoreboard, mainly for debugging
function void Scoreboard::display(input string prefix);
  $display("@%0t: %m so far %0d expected cells, %0d actual
rcvd", $time, iexpect, iactual);
  foreach (expect_cells[i]) begin
    $display("Tx[%0d]: exp=%0d, act=%0d",
            i, expect_cells[i].iexpect, expect_cells[i].iactual);
  foreach (expect_cells[i].q[j])
    expect_cells[i].q[j].display(
                    $sformatf("%sScoreboard: Tx%0d: ", prefix, i));
  end
endfunction : display
```

Sample 11.24 shows the class used to gather functional coverage. Since the coverage only looks at data in a single class, the cover group is defined and instantiated inside the `Coverage` class. The data values are read by the class's `sample()` method, then the cover group's `sample()` method is called to record the values.

Sample 11.24 Functional coverage class

```
class Coverage;
  bit [1:0] src;
  bit [NumTx-1:0] fwd;

  covergroup CG_Forward;
    coverpoint src
      {bins src[] = {[0:3]};
       option.weight = 0;}
    coverpoint fwd
      {bins fwd[] = {[1:15]}; // Ignore fwd==0
       option.weight = 0;}
    cross src, fwd;
  endgroup : CG_Forward

  function new();
    CG_Forward = new;    // Instantiate the covergroup
  endfunction : new

  // Sample input data
  function void sample(input bit [1:0] src,
                       input bit [NumTx-1:0] fwd);
    $display("@%0t: Coverage: src=%d. FWD=%b", $time, src, fwd);
    this.src = src;
    this.fwd = fwd;
    CG_Forward.sample();
  endfunction : sample
endclass : Coverage
```

Sample 11.25 shows the `CPU_driver` class that contains the methods to drive the CPU interface.

Sample 11.25 The CPU_driver class

```
class CPU_driver;
  vCPU_T mif;
  CellCfgType lookup [255:0]; // copy of look-up table
  Config cfg;
  bit [NumTx-1:0] fwd;

  extern function new(vCPU_T mif, Config cfg);
  extern task Initialize_Host ();
  extern task HostWrite (int a, CellCfgType d); // configure
  extern task HostRead (int a, output CellCfgType d);
  extern task run();
endclass : CPU_driver


function CPU_driver::new(input vCPU_T mif, Config cfg);
  this.mif = mif;
  this.cfg = cfg;
endfunction : new


task CPU_driver::Initialize_Host();
  mif.BusMode <= 1;
  mif.Addr <= 0;
  mif.DataIn <= 0;
  mif.Sel <= 1;
  mif.Rd_DS <= 1;
  mif.Wr_RW <= 1;
endtask : Initialize_Host


task CPU_driver::HostWrite (int a, CellCfgType d); // configure
  #10 mif.Addr <= a; mif.DataIn <= d; mif.Sel <= 0;
  #10 mif.Wr_RW <= 0;
  while (mif.Rdy_Dtack!==0) #10;
  #10 mif.Wr_RW <= 1; mif.Sel <= 1;
  while (mif.Rdy_Dtack==0) #10;
endtask : HostWrite


task CPU_driver::HostRead (input int a, output CellCfgType d);
  #10 mif.Addr <= a; mif.Sel <= 0;
  #10 mif.Rd_DS <= 0;
  while (mif.Rdy_Dtack!==0) #10;
  #10 d = mif.DataOut; mif.Rd_DS <= 1; mif.Sel <= 1;
  while (mif.Rdy_Dtack==0) #10;
endtask : HostRead


task CPU_driver::run();
  CellCfgType CellFwd;
  Initialize_Host();

  // Configure through Host interface
  repeat (10) @(negedge clk);
  $write("Memory: Loading ... ");
  for (int i=0; i<=255; i++) begin
    CellFwd.FWD = $urandom();
`ifdef FWDALL
    CellFwd.FWD = '1
`endif
    CellFwd.VPI = i;
    HostWrite(i, CellFwd);
    lookup[i] = CellFwd;
  end

  // Verify memory
  $write("Verifying ...");
  for (int i=0; i<=255; i++) begin
    HostRead(i, CellFwd);
    if (lookup[i] != CellFwd) begin
    $display("FATAL, Mem Loc 0x%x contains 0x%x, expected 0x%x",
                 i, CellFwd, lookup[i]);
      $finish;
    end
  end
  $display("Verified");

endtask : run
```

11.3 Alternate Tests

The simplest test program is shown in Sample 11.2 and runs with very few constraints. During verification, you will be creating many tests, depending on the major functionality to be tested. Each test can then be run with different seeds.

11.3.1 Your First Test - Just One Cell

The first test you run should probably have just one cell, such as the test in Sample 11.26. You can add a new constraint to the `Config` class by extending it, and then injecting a new object into the environment before randomization. Once this test works, you can try two cells, then rewrite the constraint on the number of cells to run longer sequences.

Sample 11.26 Test with one cell

```
program automatic test
  #(parameter int NumRx = 4, parameter int NumTx = 4)
   (Utopia.TB_Rx Rx[0:NumRx-1],
    Utopia.TB_Tx Tx[0:NumTx-1],
    cpu_ifc.Test mif,
    input logic rst, clk);

`include "environment.sv"
  Environment env;

class Config_1_cell extends Config;
  constraint one_cells {nCells == 1; }

  function new(input int NumRx,NumTx);
    super.new(NumRx,NumTx);
  endfunction : new
endclass : Config_1_cells


  initial begin
    env = new(Rx, Tx, NumRx, NumTx, mif);

    begin // Just simulate for 1 cell
      Config_1_cells c1 = new(NumRx,NumTx);
      env.cfg = c1;
    end

    env.gen_cfg();    // Config will have just 1 cell
    env.build();
    env.run();
    env.wrap_up();
  end

endprogram // test
```

11.3.2 *Randomly Drop Cells*

The next test you may run creates errors by occasionally dropping cells, as shown in Sample 11.27. You need to make a callback for the driver that sets the drop bit. Then, in the test, inject this new functionality after the driver class has been constructed during the build phase.

Sample 11.27 Test that drops cells using driver callback

```
program automatic test
  #(parameter int NumRx = 4, parameter int NumTx = 4)
   (Utopia.TB_Rx Rx[0:NumRx-1],
    Utopia.TB_Tx Tx[0:NumTx-1],
    cpu_ifc.Test mif,
    input logic rst, clk);

`include "environment.sv"
  Environment env;

class Driver_cbs_drop extends Driver_cbs;
  virtual task pre_tx(input ATM_cell cell, ref bit drop);
    // Randomly drop 1 out of every 100 transactions
    drop = ($urandom_range(0,99) == 0);
  endtask
 endclass


  initial begin
    env = new(Rx, Tx, NumRx, NumTx, mif);
    env.gen_cfg();
    env.build();

    begin             // Create error injection callback
      Driver_cbs_drop dcd = new();
      env.drv.cbs.push_back(dcd); // Put into driver's Q
    end

    env.run();
    env.wrap_up();
  end

endprogram // test
```

11.4 Conclusion

This chapter shows how you can build a layered testbench, following the guidelines in this book. You can then create new tests by just modifying a single file and injecting new behavior, utilizing the hooks such as callbacks and multiple environment phases.

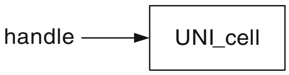
The testbench was able to get to 100% functional coverage of the ATM switch, at least for the basic cover group. You can use this example to explore more about SystemVerilog testbenches.

11.5 Exercises

1. In Sample 11.2, why is `clk` not passed into the port list of program `test`?
2. In Sample 11.6, could `numRx` be substituted for `Rx.size()` ? Why or why not?
3. For the following code snippet from Sample 11.6, explain what is being created for each statement.

```
function void Environment::build();
  cpu = new(mif, cfg);
  gen = new[numRx];
  drv = new[numRx];
  gen2drv = new[numRx];
  drv2gen = new[numRx];
  scb = new(cfg);
  cov = new();
  foreach(gen[i]) begin
    gen2drv[i] = new();
    gen[i] = new(gen2drv[i], drv2gen[i],
                 cfg.cells_per_chan[i], i);
    drv[i] = new(gen2drv[i], drv2gen[i], Rx[i], i);
 end
 ...
endfunction : build
```

4. In Sample 11.9, what coverage object does the handle `cov` point to?
5. In Sample 11.17, the function `UNI_cell::copy` assumes that the handle to the object `UNI_cell` points to an object of class `UNI_cell` as depicted in the following drawing. Draw what object the handle `dst` points to for the following function calls.



   a. `copy();`

   b. `copy(handle);`

6. In Sample 11.18, why are the `$cast()` required?
7. In Sample 11.19 and 11.20, why are the `typedef` declaration needed?
8. In Sample 11.19, why is `peek()` used first and then later a `get()`?
9. In Sample 11.23, is the error message "...`cell not found`..." in the function `check_actual` printed every time it are called? Why or why not?
10. Why do classes `Environment, Scoreboard`, and `CPU_driver` all define a handle to class `Config`? Are 3 objects of class `Config` created?