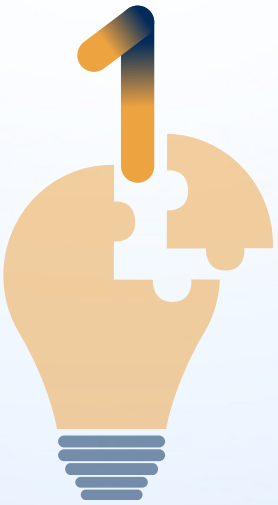
SYSTEM VERILOG TESTBENCH

EXAMPLE



JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

SystemVerilog Testbench Example 1

Let us look at a practical SystemVerilog testbench example with all the verification components and how concepts in SystemVerilog have been used to create a reusable environment.

Design

This Verilog module represents a simple register controller (reg_ctrl) with a memory element that stores data for each address. It is designed to handle read and write transactions on a memory array.

This design represents a basic memory controller, and its behavior conforms to the described protocol where write data is provided in a single clock cycle along with the address, and read data is received on the next clock cycle. Transactions can only be initiated when the ready signal is high.

```
// Note that in this protocol, write data is provided
// in a single clock along with the address while read
// data is received on the next clock, and no transactions
// can be started during that time indicated by "ready" signal.
module reg_ctrl #
(
  parameter ADDR_WIDTH = 8,
  parameter DATA_WIDTH = 16,
  parameter DEPTH = 256,
  parameter RESET_VAL = 16'h1234
)
(
  input clk,
  input rstn,
  input [ADDR_WIDTH-1:0] addr,
  input sel,
  input wr,
  input [DATA_WIDTH-1:0] wdata,
  output reg [DATA_WIDTH-1:0] rdata,
  output reg ready
);

// Some memory element to store data for each addr
reg [DATA_WIDTH-1:0] ctrl [DEPTH];
reg ready_dly;
wire ready_pe;

// If reset is asserted, clear the memory element
// Else store data to addr for valid writes
// For reads, provide read data back
always @(posedge clk)
begin
  if (!rstn)
```

```
  begin
    for (int i = 0; i < DEPTH; i += 1)
    begin
      ctrl[i] <= RESET_VAL;
    end
  end
  else
  begin
    if (sel & ready & wr)
    begin
      ctrl[addr] <= wdata;
    end
    if (sel & ready & !wr)
    begin
      rdata <= ctrl[addr];
    end
    else
    begin
      rdata <= 0;
    end
  end
end

// Ready is driven using this always block
// During reset, drive ready as 1
// Else drive ready low for a clock low
always @(posedge clk)
begin
  if (!rstn)
  begin
    ready <= 1;
  end
  else
  begin
    if (sel & ready_pe)
    begin
      ready <= 1;
    end
    if (sel & ready & !wr)
    begin
      ready <= 0;
    end
  end
end

// Drive internal signal accordingly
always @(posedge clk)
begin
  if (!rstn)
    ready_dly <= 1;
  else
    ready_dly <= ready;
end

assign ready_pe = ~ready & ready_dly;

endmodule
```

Interface

```
// The interface allows verification components to access DUT signals
// using a virtual interface handle interface reg_if (input bit clk);
logic rstn; logic [7:0] addr;
logic [15:0] wdata;
logic [15:0] rdata;
logic   wr;
logic   sel;
logic   ready;
endinterface
```

Transaction Object

```
class reg_item;
// This is the base transaction object that will be used
// in the environment to initiate new transactions and
// capture transactions at DUT interface rand bit [7:0] addr;
rand bit [15:0] wdata;
bit [15:0]  rdata;
rand bit    wr;

// This function allows us to print contents of the data packet
// so that it is easier to track in a logfile function void print(string
tag="");
$display ("T=%0t [%s] addr=0x%0h wr=%0d wdata=0x%0h rdata=0x%0h",
$time, tag, addr, wr, wdata, rdata);
endfunction
endclass
```

Driver

```
// The driver is responsible for driving transactions to the DUT
// All it does is to get a transaction from the mailbox if it is
// available and drive it out into the DUT interface.
class driver;
  virtual reg_if vif;
  event drv_done;
  mailbox drv_mbx;

  task run();
    $display("T=%0t [Driver] starting ...", $time); @ (posedge vif.clk);

    // Try to get a new transaction every time and then assign
    // packet contents to the interface. But do this only if the
    // design is ready to accept new transactions forever
    reg_item item;

    $display("T=%0t [Driver] waiting for item ...", $time);
drv_mbx.get(item);
    item.print("Driver");
```

```
    vif.sel <= 1;
    vif.addr <= item.addr;
    vif.wr <= item.wr;
    vif.wdata <= item.wdata;
    @ (posedge vif.clk);
    while (!vif.ready) begin
      $display("T=%0t [Driver] wait until ready is high", $time);
      @(posedge vif.clk);
    end

    // When transfer is over, raise the done event
    vif.sel <= 0;
    ->drv_done;
  endtask
endclass
```

Monitor

```
// The monitor has a virtual interface handle with which it can monitor
// the events happening on the interface. It sees new transactions and then
// captures information into a packet and sends it to the scoreboard
// using another mailbox.
class monitor;
  virtual reg_if vif;
  mailbox scb_mbx; // Mailbox connected to scoreboard

  task run();
    $display("T=%0t [Monitor] starting ...", $time);

    // Check forever at every clock edge to see if there is a
    // valid transaction and if yes, capture info into a class
    // object and send it to the scoreboard when the transaction
    // is over.
    forever begin
      @(posedge vif.clk);
      if (vif.sel) begin
        reg_item item = new;
        item.addr = vif.addr;
        item.wr = vif.wr;
        item.wdata = vif.wdata;
        if (!vif.wr) begin
          @(posedge vif.clk);
          item.rdata = vif.rdata;
        end
        item.print("Monitor");
        scb_mbx.put(item);
      end
    end
  endtask
endclass
```

Environment

```
// The environment is a container object simply to hold all verification
// components together. This environment can then be reused later and all
// components in it would be automatically connected and available for use
// This is an environment without a generator.
class env;
  driver d0;
  // Driver to design
  monitor m0;
  // Monitor from design
  scoreboard s0;
  // Scoreboard connected to monitor
  mailbox scb_mbx;
  // Top-level mailbox for SCB <-> MON
  virtual reg_if vif;
  // Virtual interface handle

  // Instantiate all testbench components
  function new();
    d0 = new;
    m0 = new;
    s0 = new;
    scb_mbx = new();
  endfunction

  // Assign handles and start all components so that
  // they all become active and wait for transactions to be
  // available
  virtual task run();
    d0.vif = vif;
    m0.vif = vif;
    m0.scb_mbx = scb_mbx;
    s0.scb_mbx = scb_mbx;

    fork
      s0.run();
      d0.run();
      m0.run();
    join_any
  endtask
endclass
```

Scoreboard

```
// The scoreboard is responsible to check data integrity. Since the design
// stores data it receives for each address, scoreboard helps to check if the
// same data is received when the same address is read at any later point
// in time. So the scoreboard has a "memory" element which updates it
// internally for every write operation.
class scoreboard;
  mailbox scb_mbx;
  reg_item refq[256];

  task run();
    forever begin
      reg_item item;
      scb_mbx.get(item);
      item.print("Scoreboard");

      if (item.wr) begin
        if (refq[item.addr] == null)
          refq[item.addr] = new;
        refq[item.addr] = item;
        $display("T=%0t [Scoreboard] Store addr=0x%0h wr=0x%0h data=0x%0h",
$time, item.addr, item.wr, item.wdata);
      end

      if (!item.wr) begin
        if (refq[item.addr] == null)
          if (item.rdata != 'h1234)
            $display("T=%0t [Scoreboard] ERROR! First time read, addr=0x%0h
exp=1234 act=0x%0h", $time, item.addr, item.rdata);
          else
            $display("T=%0t [Scoreboard] PASS! First time read, addr=0x%0h
exp=1234 act=0x%0h", $time, item.addr, item.rdata);
        else
          if (item.rdata != refq[item.addr].wdata)
            $display("T=%0t [Scoreboard] ERROR! addr=0x%0h exp=0x%0h
act=0x%0h", $time, item.addr, refq[item.addr].wdata, item.rdata);
          else
            $display("T=%0t [Scoreboard] PASS! addr=0x%0h exp=0x%0h
act=0x%0h", $time, item.addr, refq[item.addr].wdata, item.rdata);
      end
    end
  endtask
endclass
```

Test

```
// an environment without the generator and hence the stimulus should be
// written in the test.
class test;
  env e0;
  mailbox drv_mbx;

  function new();
    drv_mbx = new();
    e0 = new();
  endfunction

  virtual task run();
    e0.d0.drv_mbx = drv_mbx;
    fork
      e0.run();
    join_none
    apply_stim();
  endtask

  virtual task apply_stim();
    reg_item item;
    $display("T=%0t [Test] Starting stimulus ...", $time);
    item = new;
    item.randomize() with { addr == 8'hAA; wr == 1; };
    drv_mbx.put(item);
    item = new;
    item.randomize() with { addr == 8'hAA; wr == 0; };
    drv_mbx.put(item);
  endtask
endclass
```

Testbench TOP

```
// Top-level testbench contains the interface, DUT, and test handles which
// can be used to start test components once the DUT comes out of reset. Or
// the reset can also be a part of the test class in which case all you need
// to do is start the test's run method.
module tb;
  reg clk;
  always #10 clk = ~clk;

  reg_if _if (clk);
  reg_ctrl u0 (
    .clk(clk),
    .addr(_if.addr),
    .rstn(_if.rstn),
    .sel(_if.sel),
    .wr(_if.wr),
    .wdata(_if.wdata),
    .rdata(_if.rdata),
    .ready(_if.ready)
  );
```

```
  initial begin
    new_test t0;
    clk <= 0;
    _if.rstn <= 0;
    _if.sel <= 0;
    #20 _if.rstn <= 1;
    t0 = new;
    t0.e0.vif = _if;
    t0.run();
    // Once the main stimulus is over, wait for some time
    // until all transactions are finished and then end
    // simulation. Note that $finish is required because
    // there are components that are running forever in
    // the background like clk, monitor, driver, etc
    #200 $finish;
  end

  // Simulator-dependent system tasks that can be used to
  // dump simulation waves.
  initial begin
    $dumpvars;
    $dumpfile("dump.vcd");
  end
endmodule
```