Memory Controller Verification: A Project-Based Guide Using SystemVerilog, OOP, and UVM Principles

Pratik Uttam Mhasawade
AElectronics and Telecommunication
VPKBIET
Baramati, India
pratik.mhasawade.entc.2023@vpkbiet.org

Aniket Appaso Saste
Electronics and Telecommunication
VPKBIET
Baramati, India
aniket.saste.entc.2023@vpkbiet.org

Gautrav Arun Shinde
Electronics and Telecommunication
VPKBIET
Baramati, India
gaurav.shinde.entc.2023@vpkbiet.org

Abstract—This paper presents a complete, expert-level SystemVerilog project for the functional verification of a single-port Random Access Memory (RAM) module. It details the construction of a layered, class-based testbench founded on Object-Oriented Programming (OOP) principles and employing Constrained Random Verification (CRV) for intelligent stimulus generation. The architecture deliberately mirrors the foundational concepts of the Universal Verification Methodology (UVM), offering a practical, "UVM-Lite" framework. Key components, including a transaction class, generator, driver, monitor, and a self-checking scoreboard, are implemented from first principles. The paper serves as a comprehensive guide for understanding the data flows, component interactions, and advanced verification techniques that are standard practice in modern System-on-Chip (SoC) design verification, providing a stepping stone for engineers transitioning to the full UVM library.

Index Terms—SystemVerilog, UVM, Functional Verification, OOP, Memory Controller, Constrained Random Verification, Testbench Automation

I. Introduction

In modern System-on-Chip (SoC) design, the functional correctness of memory subsystems is paramount to system reliability [1]. As design complexity has grown, traditional directed testing methods have become insufficient for achieving comprehensive verification coverage. This has driven the adoption of advanced, methodology-driven verification techniques that provide greater automation, efficiency, and bug-finding capabilities [2].

This paper details a verification environment for a simple RAM module, built using SystemVerilog. The architecture embodies the principles of the Universal Verification Methodology (UVM) and is constructed as a layered testbench [3]. This paradigm promotes reusability and scalability by separating distinct verification concerns into independent, encapsulated components:

- Stimulus Generation: Creates high-level, abstract data operations (transactions).
- Driving: Translates abstract transactions into signal-level activity for the Design Under Test (DUT).
- Monitoring: Observes DUT signal activity and converts it back into abstract transactions.
- Checking: Compares actual results against a reference model to verify correctness.

This separation ensures that a change in one area, such as the DUT's pin-level protocol, only requires localized updates, leaving the stimulus and checking logic intact. The core of our stimulus strategy is Constrained Random Verification (CRV), which shifts the engineering effort from writing tedious test vectors to formally describing the system's intended behavior through constraints [4]. This approach excels at uncovering unexpected corner-case bugs and enhances productivity [5].

II. The Design Under Test (DUT)

The DUT for this project is a parameterized, synchronous, single-port RAM, a fundamental building block in digital design.

A. Architectural Details

The module is defined with DATA_WIDTH and ADDR_WIDTH parameters, allowing the same description to be instantiated with various configurations, a key principle of reusable IP design. All operations are synchronized to the positive edge of the clock. The read operation models a registered output, where data is available on the clock cycle following the read request, a common behavior for on-chip block RAMs.

TABLE I
Port Descriptions for simple_ram DUT

Port Name	Direction	Description
clk	input	Master clock signal.
rst_n	input	Active-low asynchronous reset.
cs	input	Chip Select; enables the port.
we	input	Write Enable (1=Write, 0=Read).
addr	input	Address bus (ADDR_WIDTH wide).
wdata	input	Write data bus (DATA_WIDTH wide).
rdata	output	Read data bus (DATA_WIDTH wide).

B. SystemVerilog RTL Code

The complete, synthesizable code for the DUT is provided below.

```
module simple_ram
#(
  parameter DATA_WIDTH = 8,
  parameter ADDR_WIDTH = 8
)
(
  input  logic clk, rst_n, cs, we,
  input  logic [ADDR_WIDTH-1:0] addr,
  input  logic [DATA_WIDTH-1:0] wdata,
  output logic [DATA_WIDTH-1:0] rdata
);

  localparam DEPTH = 2**ADDR_WIDTH;
  logic [DATA_WIDTH-1:0] mem [DEPTH-1:0];

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      for (int i = 0; i < DEPTH; i++) begin
        mem[i] <= '0;
      end
      rdata <= '0;
    end
    else begin
      if (cs) begin
        if (we) begin
          mem[addr] <= wdata;
        end
        rdata <= mem[addr];
      end
    end
  end
endmodule
```

Listing 1. The simple_ram DUT module.

III. The Object-Oriented Verification Environment

Our verification environment is composed of interacting SystemVerilog classes, each encapsulating specific functionality. Fig. 1 illustrates the high-level architecture.

Verification Architecture Blueprint

$$\begin{tikzcd}\text{Generator} \arrow[r, "\text{Mailbox}"] & \text{Driver} \arrow[r, "\text{VIF}"] & \text{DUT}\end{tikzcd}$$

$$\begin{tikzcd}\text{Monitor} \arrow[r, "\text{VIF}"] & \text{DUT}\end{tikzcd}$$

$$\begin{tikzcd}\text{Monitor} \arrow[r, "\text{Mailbox}"] & \text{Scoreboard}\end{tikzcd}$$

Fig. 1. High-level architecture of the layered testbench, showing communication pathways between components.

A. Transaction Class (mem_transaction)

The transaction is the fundamental data packet, abstracting a single DUT operation. Its properties are declared rand to enable CRV.

```
class mem_transaction;
  rand bit [7:0] addr;
  rand bit [7:0] wdata;
  rand bit       we;
  bit [7:0] rdata;

  // 60% writes, 40% reads
  constraint wr_rd_dist_c {
    we dist {1 := 6, 0 := 4};
  }

  function void print(string tag = ”TRANS”);
    $display(”[%s] @ %0t: Addr=0x%0h, WE=%b, WData=0x%0h, RData=0x%0h”,
             tag, $time, addr, we, wdata, rdata);
  endfunction
endclass
```

Listing 2. The mem_transaction class.

The wr_rd_dist_c constraint uses a weighted distribution to steer randomization, making write operations more frequent than reads [6].

B. Generator

The generator creates and randomizes mem_transaction objects, sending them to the driver via a mailbox, a synchronized message queue that decouples the producer from the consumer.

```
class generator;
  mailbox #(mem_transaction) gen2drv_mbx;
  int repeat_count;

  function new(mailbox #(mem_transaction) gen2drv_mbx, int repeat_count = 10);
    this.gen2drv_mbx  = gen2drv_mbx;
    this.repeat_count = repeat_count;
  endfunction

  task run();
    mem_transaction tx;
    repeat (repeat_count) begin
      tx = new();
      if (!tx.randomize())
        $fatal(1, ”Randomization failed!”);
      tx.print(”GEN SEND”);
      gen2drv_mbx.put(tx);
    end
  endtask
endclass
```

Listing 3. The generator class.

C. Driver

The driver bridges the transaction-level and signal-level domains. It receives transactions and drives the DUT's pins according to the memory protocol. Connection to the static DUT is achieved via a virtual interface, a key construct for linking dynamic classes to static hardware [7].

```
class driver;
  mailbox #(mem_transaction) gen2drv_mbx;
  virtual mem_if vif;

  function new(mailbox #(mem_transaction) gen2drv_mbx);
    this.gen2drv_mbx = gen2drv_mbx;
  endfunction

  task run();
    forever begin
      mem_transaction tx;
      gen2drv_mbx.get(tx); // Blocking call
      tx.print(”DRV RECV”);

      @(posedge vif.clk);
       vif.cs <= 1;
       vif.we <= tx.we;
       vif.addr <= tx.addr;
       if (tx.we) vif.wdata <= tx.wdata;

      @(posedge vif.clk);
       vif.cs <= 0;
       vif.wdata <= '0;
    end
  endtask
endclass
```

Listing 4. The driver class.

D. Monitor

The monitor passively observes DUT signals, reconstructs transactions, and sends them to the scoreboard. It is the testbench's source of truth regarding DUT activity.

```
class monitor;
  mailbox #(mem_transaction) mon2scb_mbx;
  virtual mem_if vif;

  function new(mailbox #(mem_transaction)
    mon2scb_mbx);
    this.mon2scb_mbx = mon2scb_mbx;
  endfunction

  task run();
    mem_transaction tx;
    forever begin
      @(posedge vif.clk);
      if (vif.cs) begin
        tx = new();
        tx.addr = vif.addr;
        tx.we = vif.we;
        if (tx.we) tx.wdata = vif.wdata;
        else begin // Handle read latency
          @(posedge vif.clk);
          tx.rdata = vif.rdata;
        end
        tx.print("MON CAPTURE");
        mon2scb_mbx.put(tx);
      end
    end
  endtask
endclass
```

Listing 5. The monitor class.

E. Scoreboard

The scoreboard is the arbiter of correctness. It contains a behavioral reference model of the DUT and compares observed transactions from the monitor against it. This automated self-checking mechanism is essential for running large-scale random regressions.

```
class scoreboard;
  mailbox #(mem_transaction) mon2scb_mbx;
  // Golden reference model
  bit [7:0] ref_mem [bit [7:0]];
  int pass_count = 0, fail_count = 0;

  function new(mailbox #(mem_transaction)
    mon2scb_mbx);
    this.mon2scb_mbx = mon2scb_mbx;
  endfunction

  task run();
    mem_transaction tx;
    forever begin
      mon2scb_mbx.get(tx);
      if (tx.we) begin // Write
        ref_mem[tx.addr] = tx.wdata;
      end else begin // Read
        if (ref_mem.exists(tx.addr)) begin
          if (tx.rdata == ref_mem[tx.addr])
            pass_count++;
          else
            $error("FAIL: Data mismatch. Addr=0x%0h,
 RData=0x%0h, Expected=0x%0h", tx.addr, tx.rdata
, ref_mem[tx.addr]);
            fail_count++;
        end else begin // Uninitialized read
          if (tx.rdata == 8'h00)
            pass_count++;
          else
            $error("FAIL: Default value mismatch.
Addr=0x%0h, RData=0x%0h, Expected=0x00", tx.addr
, tx.rdata);
            fail_count++;
        end
      end
    end
  endtask
endclass
```

Listing 6. The scoreboard class.

The reference model is implemented with an associative array, which is ideal for modeling sparse memories.

F. Environment

The environment class is a container that instantiates and connects all verification components, making the entire testbench portable and manageable.

```
class environment;
  generator  gen;
  driver     drv;
  monitor    mon;
  scoreboard scb;

  mailbox #(mem_transaction) gen2drv_mbx;
  mailbox #(mem_transaction) mon2scb_mbx;
  virtual mem_if vif;

  function new(virtual mem_if vif);
    this.vif = vif;
    gen2drv_mbx = new();
    mon2scb_mbx = new();
    gen = new(gen2drv_mbx, 50); // Gen 50 txns
    drv = new(gen2drv_mbx);
    mon = new(mon2scb_mbx);
    scb = new(mon2scb_mbx);
  endfunction

  task run();
    drv.vif = this.vif;
    mon.vif = this.vif;
    // Start all components concurrently
    fork
      gen.run();
      drv.run();
      mon.run();
      scb.run();
    join_none
  endtask
endclass
```

Listing 7. The environment class.

IV. Top-Level Integration and Simulation

The top-level testbench is a static module that instantiates the DUT, the verification environment, and the interface that connects them.

A. SystemVerilog Interface (mem_if)

An interface encapsulates the DUT signals, simplifying connections and reducing errors.

```
interface mem_if(input logic clk, input logic rst_n)
    ;
  parameter DATA_WIDTH = 8;
  parameter ADDR_WIDTH = 8;
  // Match DUT signals
  logic cs;
  logic we;
  logic [ADDR_WIDTH-1:0] addr;
  logic [DATA_WIDTH-1:0] wdata;
  logic [DATA_WIDTH-1:0] rdata;
endinterface
```

Listing 8. The mem_if interface.

B. Top-Level Testbench (test_top)

The test_top module is the root of the simulation. It generates the clock and reset, instantiates the DUT and interface, creates the environment class, and starts the test.

```
module test_top;
  logic clk;
  logic rst_n;

  // Clock and reset generation
  always #5 clk = ~clk;
  initial begin
    clk <= 0; rst_n <= 0;
    #20; rst_n <= 1;
  end

  // Instantiate interface and DUT
  mem_if #(8, 8) vif(clk, rst_n);
  simple_ram #(8, 8) dut (
    .clk(vif.clk), .rst_n(vif.rst_n),
    .cs(vif.cs),   .we(vif.we),
    .addr(vif.addr), .wdata(vif.wdata),
    .rdata(vif.rdata)
  );

  // Instantiate and run the environment
  initial begin
    environment env;
    env = new(vif); // Pass VIF handle
    env.run();
    // A simplified wait for test completion
    #5000;
    $display("Final Score: %0d Passed, %0d Failed.",
             env.scb.pass_count, env.scb.fail_count)
    ;
    $finish;
  end

  // Waveform dumping
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, test_top);
  end
endmodule
```

Listing 9. The test_top module.

The initial block that instantiates the environment is the critical link; it passes the handle to the physical interface (‘vif‘) to the environment’s constructor, establishing the connection to the dynamic, class-based testbench. System tasks ‘*dumpfile*‘*and*‘dumpvars‘ are included to generate a VCD waveform file for debugging.

V. Conclusion

This paper has detailed the design and implementation of a complete, UVM-inspired verification environment for a single-port RAM using SystemVerilog and object-oriented principles. By building each component—transaction, generator, driver, monitor, and scoreboard—from the ground up, we have demonstrated the core tenets of a modern, layered testbench architecture. The use of Constrained Random Verification and a self-checking scoreboard highlights the power of automation in finding bugs and achieving verification closure. This project serves as a practical, hands-on learning tool that bridges the gap between basic SystemVerilog knowledge and the industrial application of comprehensive verification methodologies like UVM.

References

[1] J. Bergeron, Writing Testbenches: Functional Verification of HDL Models, 2nd ed. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[2] K. D. Tasiran, S. Keckler, ”The limits of directed testing,” in Proceedings of the 38th Design Automation Conference (DAC ’01), 2001, pp. 22-27.

[3] S. Spear and G. Tumbush, SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, 3rd ed. New York, NY, USA: Springer, 2012.

[4] T. Fitzpatrick and A. Rich, ”It’s the methodology, stupid!” in Proceedings of the 38th Design Automation Conference (DAC ’01), 2001, pp. 624-627.

[5] C. J. Brej, J. L. Analoui, ”Coverage-driven verification,” in Proceedings of the IEEE International Conference on Computer Design (ICCD ’04), 2004, pp. 128-133.

[6] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017, 2017.

[7] R. G. Kumar, ”SystemVerilog Virtual Interfaces for Reusable Verification Environments,” in SNUG San Jose, 2006.