SystemVerilog and UVM

Introduction

- SystemVerilog is an extension of Verilog, designed to enhance verification capabilities.
- It supports both hardware description (RTL) and testbench development.
- SystemVerilog combines features for both design and verification, making it a versatile language in the field of digital system development.

A brief history of SystemVerilog

1. ****Verilog Introduction (1984):**
   - Verilog, the predecessor of SystemVerilog, was introduced by Gateway Design Automation (later acquired by Cadence Design Systems) as a hardware description language for modeling and simulating digital circuits.
2. **IEEE Standardization (1995):**
   - The Institute of Electrical and Electronics Engineers (IEEE) standardized Verilog as IEEE 1364-1995. This version provided a standardized syntax for Verilog, making it more widely adopted in the industry.
3. **Accellera Formation (2000):**
   - The need for advanced verification capabilities led to the formation of Accellera, an industry consortium focused on electronic design automation (EDA) standards. Accellera started working on extensions to Verilog for better verification methodologies.

4. **SystemVerilog Emergence (2002):**
    - The SystemVerilog language was introduced to the industry as an extension of Verilog. It aimed to address the growing challenges in digital design and verification by incorporating features for advanced verification and design description.
5. **IEEE Standardization of SystemVerilog (2005):**
    - IEEE standardized SystemVerilog as IEEE 1800-2005. This marked the official recognition of SystemVerilog as a distinct language with capabilities beyond traditional Verilog, including advanced verification features.
6. **UVM Development (2009):**
    - The Universal Verification Methodology (UVM) was developed on top of SystemVerilog to provide a standardized methodology for verification. UVM aimed to improve the efficiency and effectiveness of verification processes through a common framework.

Key Features of SystemVerilog

- **Enhancement of Verilog:** SystemVerilog builds upon the foundation of Verilog, incorporating new features to improve expressiveness and usability.
- **Hardware Description:** Like Verilog, SystemVerilog is used for describing digital hardware at various levels of abstraction, such as Register Transfer Level (RTL) and Gate Level.
- **Verification Capabilities:** One of the significant advancements in SystemVerilog is its robust support for verification. It provides constructs for creating powerful and scalable testbenches.
- **Object-Oriented Programming (OOP):** SystemVerilog introduces object-oriented programming features, including classes, objects, inheritance, and polymorphism. This makes it well-suited for creating modular and reusable verification environments.
- **Constrained Random Verification:** SystemVerilog facilitates constrained random testing, allowing the generation of realistic and diverse test scenarios. This is particularly useful in verification environments to catch corner-case scenarios.

UVM (Universal Verification Methodology)

UVM is a widely adopted verification methodology built on top of SystemVerilog. Studying SystemVerilog is crucial for understanding and implementing UVM-based testbenches, which are prevalent in the semiconductor industry.

Datatypes

Uses both 2-state and 4-state Data Types:

2-state Data Types (0,1)	
bit	User defined vector size. Default unsigned
byte	8-bit signed integer
shortint	16-bit signed integer
int	32-bit signed integer
longint	64-bit signed integer
4-state Data Types (0, 1, x, z)	
logic (**reg**)	User defined vector size. Default unsigned
integer	32-bit signed integer
time	64-bit signed integer

Example: Verilog vs. SystemVerilog

```
// Verilog module construct
module Verilog (clk, enable, data, out);
        // Input port declaration
        input           clk, enable;
        input  [2:0]    data;
        // Output port declaration
        output          out;
        // Optional data type
        //              declaration
        wire            clk, reset, enable;
        reg             out;

        always @ (posedge clk)
        begin
          if (enable)
            out <= /*do something*/;
        end
endmodule
```

```
// SystemVerilog module construct
module SysVerilog (input   clk, enable,
                   input  [2:0] data,
                   output       out);
        // Data type always default to
        //              type logic

        always_ff @ (posedge clk)
        begin
          if (enable)
            out <= /*do something*/;
        end
endmodule
```

```
// CounterClass.sv

class Counter;
    // Properties
    int count;

    // Constructor
    function new();
        count = 0; // Initialize count to zero
    endfunction

    // Methods
    function void increment();
        count = count + 1;
    endfunction

    function void decrement();
        if (count > 0)
            count = count - 1;
    endfunction

    function int getValue();
        return count;
    endfunction
endclass
```

Class in SV

```
module Testbench;
    // Instantiate the Counter class
    Counter myCounter;

    initial begin
        // Create an instance of the Counter class
        myCounter = new();

        // Access and use class methods
        myCounter.increment();
        myCounter.increment();
        myCounter.decrement();

        // Display the current count value
        $display("Current Count: %0d", myCounter.getValue());
    end
endmodule
```

Task vs Function

There are two main differences between functions and tasks.

- When we write a SystemVerilog function, it performs a calculation and returns a single value.
- In contrast, a SystemVerilog task executes a number of sequential statements but doesn't return a value. Instead, the task can have an unlimited number of outputs .
- In addition to this, SystemVerilog functions execute immediately and can't contain time consuming constructs such as delays, posedge macros or wait statements.
- In contrast, we can use time consuming constructs inside of a SystemVerilog task.

```
function int addNumbers(int a, int b);
    return a + b;
endfunction

// Function invocation
int result = addNumbers(3, 5);
```

```
task printMessage(string msg);
    $display("Message: %s", msg);
endtask

// Task invocation
initial begin
    printMessage("Hello,
SystemVerilog!");
end
```

Inheritance

SystemVerilog, inheritance is a feature of object-oriented programming (OOP) that allows a class to inherit properties and methods from another class. SystemVerilog supports single inheritance, meaning a class can inherit from at most one parent class. This feature promotes code reuse, modularity, and flexibility in designing complex systems.

```
class parent_trans;
  bit [31:0] data;

  function void disp_p();
    $display("Value of data = %0h", data);
  endfunction
endclass

class child_trans extends parent_trans;
  int id;

  function void disp_c();
    $display("Value of id = %0h", id);
  endfunction
endclass
```

```
module class_example;
  initial begin
    child_trans c_tr;
    c_tr = new();
    c_tr.data = 5; // child class is updating property of its base class
    c_tr.id = 1;

    c_tr.disp_p(); // child class is accessing method of its base class
    c_tr.disp_c();
  end
endmodule
```

SystemVerilog Polymorphism

Polymorphism means having many forms. A base class handle can invoke methods of its child class which has the same name. Hence, an object can take many forms.

1. As we know, the derived class object can override methods of its base class. Similarly, the base class object can also override the method of one of the child classes. It means a base class method has different forms based on derived class implementation.
2. To use many forms of the method, the virtual keyword must be used in the method definition.

```
class parent;
  bit [31:0] data;
  int id;

  virtual function void display();
     $display("Base: Value of data = %0d, id = %0d", data, id);
  endfunction
endclass

class child_A extends parent;
  function void display();
    $display("Child_A: Value of data = %0d, id = %0d", data, id);
  endfunction
endclass

class child_B extends parent;
  function void display();
    $display("Child_B: Value of data = %0d, id = %0d", data, id);
  endfunction
endclass
```

```
class child_C extends parent;
  function void display();
    $display("Child_C: Value of data = %0d, id = %0d", data, id);
  endfunction
endclass
```

```
module class_example;
  initial begin
    parent p_A, p_B, p_C;
    child_A c_A = new();
    child_B c_B = new();
    child_C c_C = new();

    c_A.data = 200;
    c_A.id   = 2;

    c_B.data = 300;
    c_B.id   = 3;

    c_C.data = 400;
    c_C.id   = 4;

    p_A = c_A;
    p_B = c_B;
    p_C = c_C;

    p_A.data = 100;
    p_A.id   = 1;

    p_A.display();
    p_B.display();
    p_C.display();
  end
endmodule
```

Output:

```
Child_A: Value of data = 100, id = 1
Child_B: Value of data = 300, id = 3
Child_C: Value of data = 400, id = 4
```

A **layered testbench architecture** is an organized and modular approach to designing the verification environment for digital designs. This architecture aims to improve code maintainability, scalability, and reusability. It involves breaking down the testbench into different layers, each with a specific responsibility. Here's a typical layered testbench architecture:

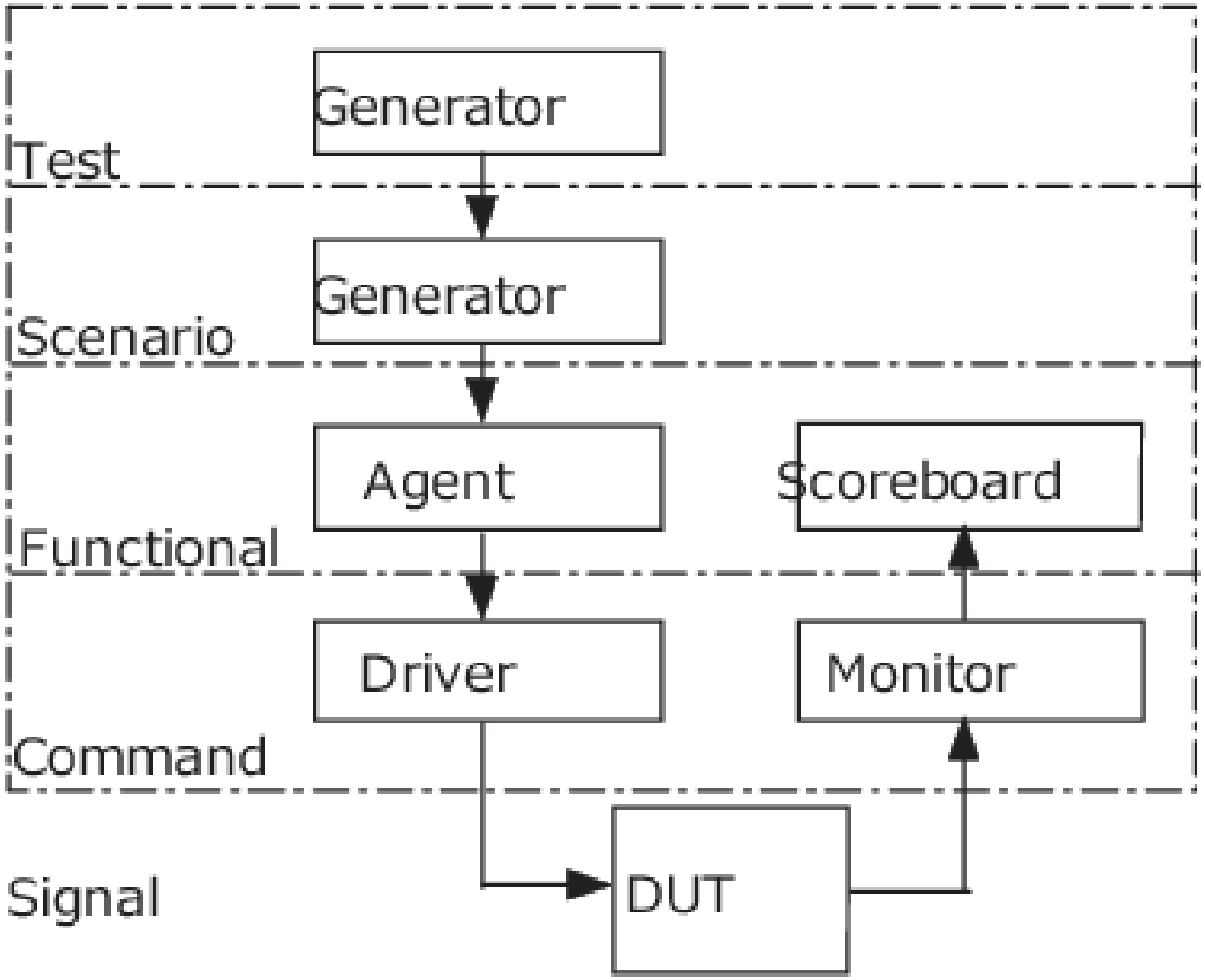
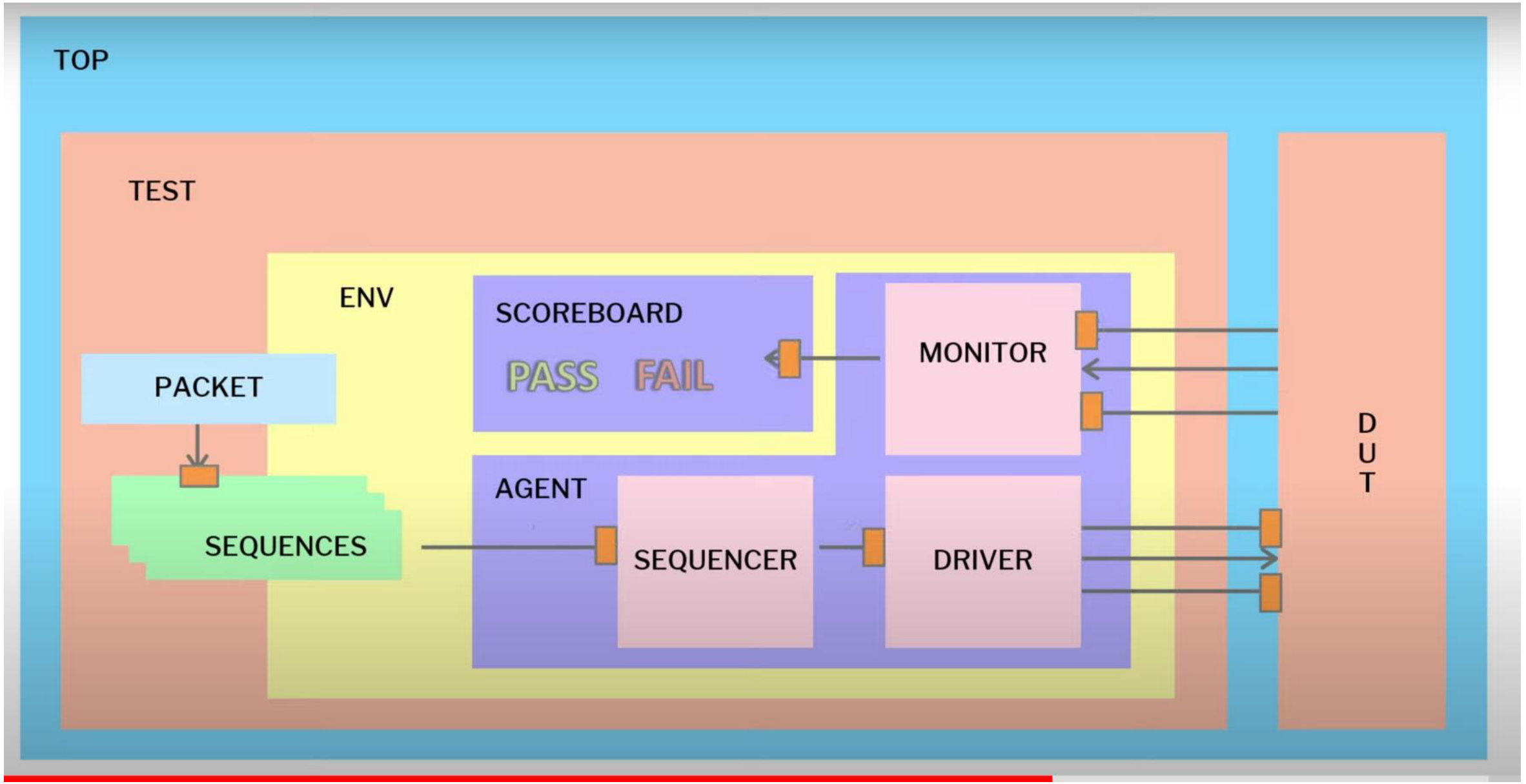


Figure 1. Layered Testbench

UVM


TOP
TEST
ENV
SCOREBOARD
PASS
FAIL
MONITOR
PACKET
AGENT
SEQUENCES
SEQUENCER
DRIVER
DUT

```
`include "umv_macros.svh"

import uvm_pkg::*;

module top();

    our_design uut(); //instantiated our design

    initial begin
        run_test("our_test");
    end

endmodule
```

```
class our_test extends uvm_test;

    `uvm_component_utils(our_test)

    //constructor
    function new ( string name = "our_test", uvm_component parent = null);
        super.new(name, parent);

    endfunction

    //main logic


    //methods
    //properties


endclass
```

```
class our_env extends uvm_env;

    `uvm_component_utils(our_env)

    function new (string name = "our_env", uvm_component parent = null);
        super.new(name, parent);

    endfunction

endclass: our_env
```

```
class our_agent extends uvm_agent;

    `uvm_component_utils(our_agent)

    function new (string name = "our_agent", uvm_component parent =
null);
        super.new(name, parent);

    endfunction

endclass: our_agent
```

```
 9 class our_sequencer extends uvm_sequencer
10 //#(sequence_item_template);
11
12     `uvm_component_utils(our_sequencer)
13
14     function new(string name =
   "our_sequencer", uvm_component parent=null);
15         super.new(name, parent);
16     endfunction: new
17
18     //main logic
19
20 endclass: our_sequencer
```

```
 9 class our_driver extends uvm_driver
10 //#(sequence_item_template);
11
12     `uvm_component_utils(our_driver)
13
14     function new(string name = "our_driver",
   uvm_component parent=null);
15         super.new(name, parent);
16     endfunction: new
17
18
19     //main logic
20
21
```

```
 9 class our_monitor extends uvm_monitor;
10
11     `uvm_component_utils(our_monitor)
12
13     function new(string name = "our_monitor",
   uvm_component parent=null);
14         super.new(name, parent);
15     endfunction: new
16
17     //main logic
18
19 endclass: our_monitor
~
~
```