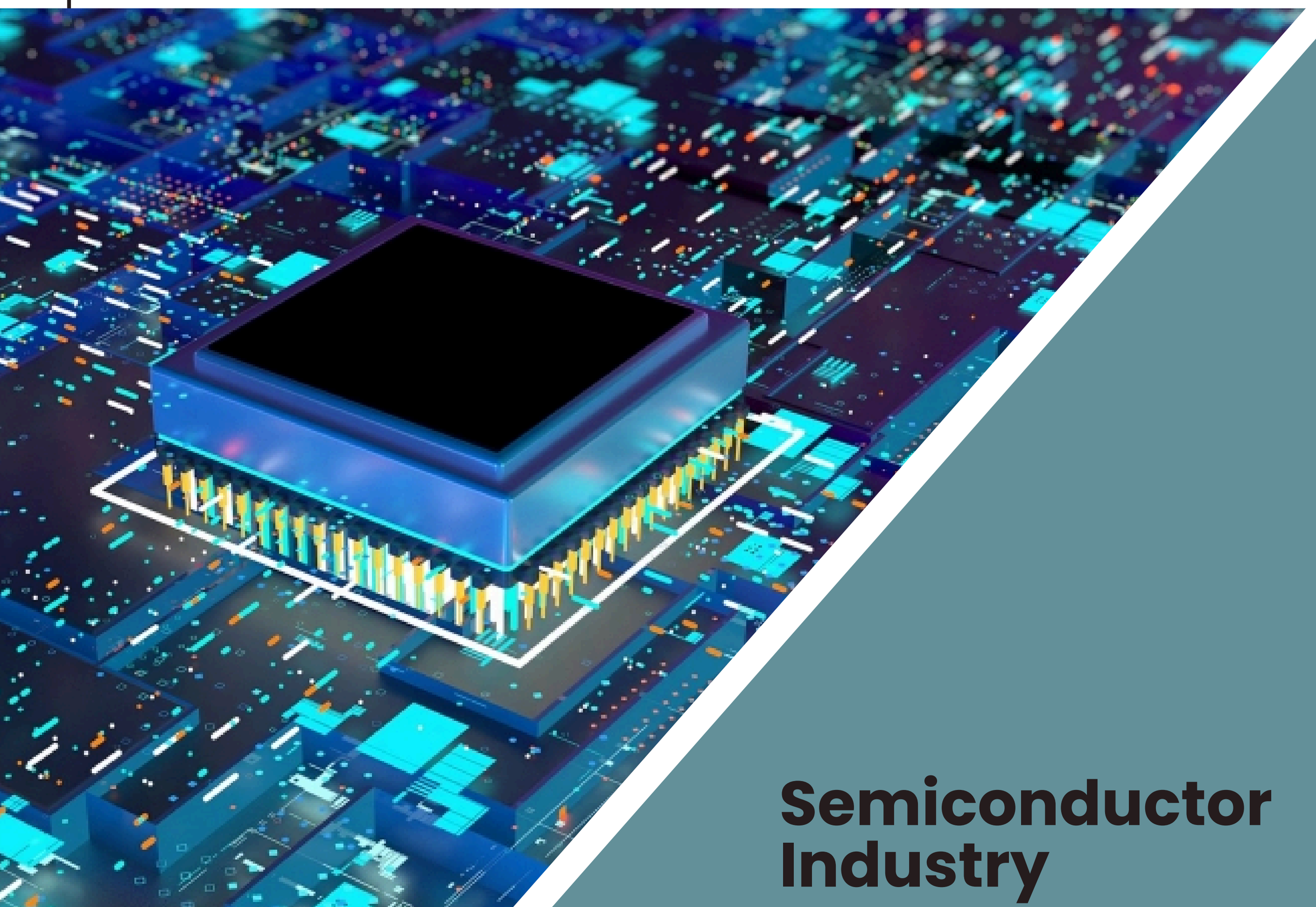
RTL TO GDS

Article-23

UVM Basics

(For Design Verification)



Semiconductor Industry

Written By-

S. Chinna Venkata Narayana Reddy

"UVM: Introduction"

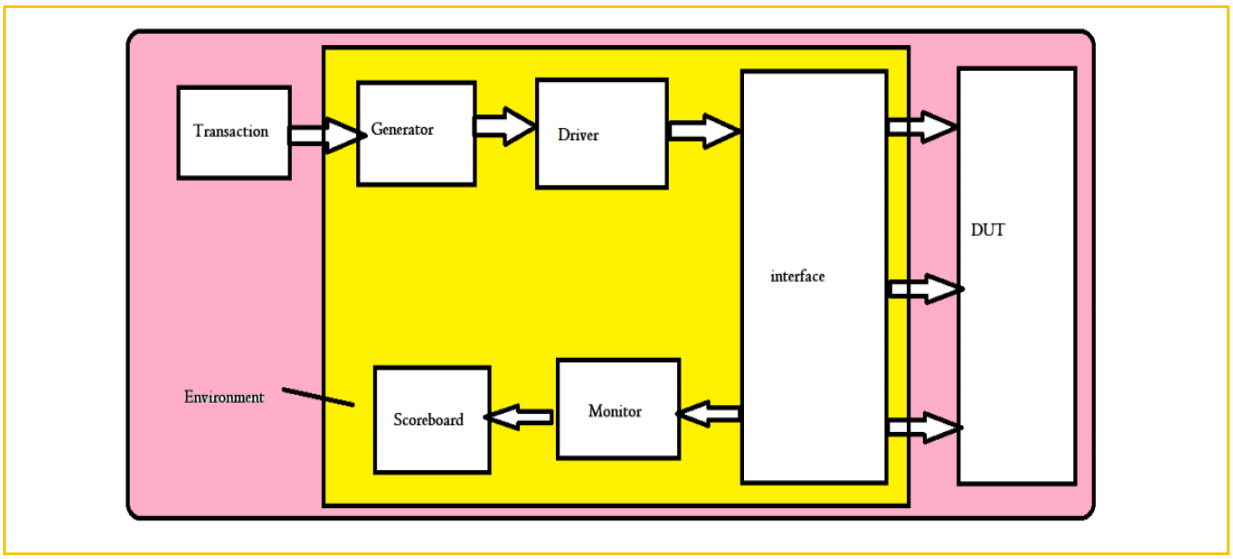
"Introduction":-

- The Accellera Universal Verification Methodology (UVM) standard verification methodology includes a set of class libraries for developing a verification environment.
- UVM is based on Open Verification Methodology (OVM) and Verification Methodology Manual (VVM).
- In System Verilog we build our verification code from Scratch but in UVM we do have some base classes so that we can extend them and build our verification environment
- The UVM API (Application Programming Interface) provides standardization for integration and creation of verification components.

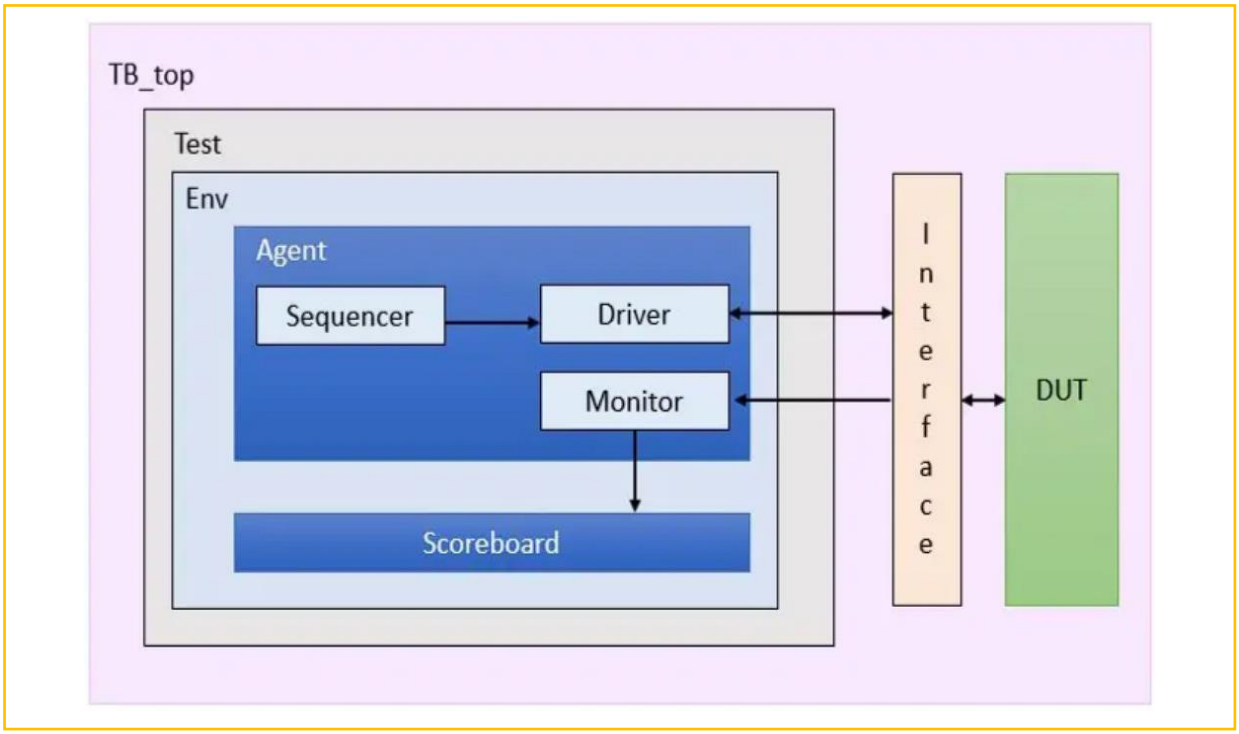
"Advantages of UVM":-

- UVM methodology provides scalable, reusable, and interoperable testbench development.
- To have uniformity in the testbench structure across the verification team, UVM provides guidelines for testbench development.
- UVM provides base class libraries so that users can inherit them to use inbuilt functionality.
- The driver-sequencer communication mechanism is an inbuilt mechanism in UVM that reduces verification efforts for the connection.
- UVM also provides verbosity to control message displays.

System Verilog TB:-



UVM TB:-



- These are two mandatory lines to start the uvm code in a console

```
`include "uvm_macros.svh"
import uvm_pkg::*;
```

"Hello World Program":-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
  initial begin
    `uvm_info("Chinnu","Hello World",UVM_MEDIUM);
  end
endmodule
```

“UVM: Reporting Macros”

“Introduction”:-

In Universal Verification Methodology (UVM), reporting macros are used to generate messages or log information during simulation. These macros are defined in the **“uvm_macros.svh”** file, and they provide a convenient way to report information at different levels of verbosity.

“Diffrenent UVM Macros”:-

- **UVM_INFO:-**Informative Messaage
- **UVM_WARNING:-**Indicates a potential problem
- **UVM_ERROR:-**Indicates a real problem.simulation continues subject to the configured message action
- **UVM_FATAL:-**Indicates a critical problem from which the simulation cannot recover. Simulation exists via $finish after a #0 delay.

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
initial begin
   `uvm_info("TB_TOP", "This is Informative message", UVM_MEDIUM)
    #10;
   `uvm_warning("TB_TOP", "This is Warning")
    #10;
   `uvm_error("TB_TOP", "This is error")
    #10;
   `uvm_fatal("TB_TOP", "This is fatal error, stopping simulation")
end
endmodule
```

“Result”:-

```
UVM_INFO /home/runner/testbench.sv(5) @ 0: reporter [TB_TOP] This is Informative message
UVM_WARNING /home/runner/testbench.sv(7) @ 10: reporter [TB_TOP] This is Warning
UVM_ERROR /home/runner/testbench.sv(9) @ 20: reporter [TB_TOP] This is error
UVM_FATAL /home/runner/testbench.sv(11) @ 30: reporter [TB_TOP] This is fatal error, stopping simulation
UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_report_server.svh(869) @ 30: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    2
UVM_WARNING :    1
UVM_ERROR :    1
UVM_FATAL :    1
** Report counts by id
[TB_TOP]     4
[UVM/RELNOTES]     1
```

“display() Vs `uvm_info”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
initial begin
   `uvm_info("TB_TOP","uvm_info", UVM_MEDIUM)
    #10;
  $display("display");#10;
end
endmodule
```

- **UVM_Info :-**
  - It gives the type of the reporting macro
  - It gives the specific file that is sending
  - It gives the line where uvm_info mentioned
  - It gives the time at which it is triggered
  - It gives the path of the component in a hierarchy
  - It gives the Id and message
- **Display:-**
  - It gives the only message
- If you want to get all the things like in uvm_info Then you need to give them manually

“Working With Verbosity”:-

- Fundamentally the Verbosity level describes how verbose a Testbench can be.
- The default Verbosity is UVM_MEDIUM.
- Different Verbosity levels are being supported by UVM.
- These are UVM_NONE, UVM_LOW, UVM_MEDIUM (Default), UVM_HIGH, UVM_FULL, UVM_DEBUG

Verbosity Level	Value
UVM_NONE	0
UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500

- To get the verbosity level we should“uvm_top.get_report_ verbosity _level()”

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
  int verbosity;
  initial begin
    verbosity = uvm_top.get_report_verbosity_level();
    $display("default Verbosity is %0d", verbosity);
    #10;
  end
endmodule;
```

“Result”:-

```
default Verbosity is 200
Simulation has finished. There are no more test vectors to simulate.
```

- The message with a verbosity level equal to or less than that threshold will be printed on the console otherwise they can't

"Code Practising":-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
  int verbosity;
  initial begin
    verbosity = uvm_top.get_report_verbosity_level();
    $display("default Verbosity is %0d", verbosity);
    #10;
    `uvm_info("Chinnu","Hello",UVM_LOW);#10;
  end
endmodule;
```

"Result":-

```
default Verbosity is 200
UVM_INFO /home/runner/testbench.sv(9) @ 10: reporter [Chinnu] Hello
Simulation has finished. There are no more test vectors to simulate.
```

- We cant see the message in the console Because the verbosity level is greater than the default verbosity value.

"Code Practising":-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
  int verbosity;
  initial begin
    verbosity = uvm_top.get_report_verbosity_level();
    $display("default Verbosity is %0d", verbosity);
    #10;
    `uvm_info("Chinnu","Hello",UVM_HIGH);#10;
  end
endmodule;
```

"Result":-

```
default Verbosity is 200
Simulation has finished. There are no more test vectors to simulate.
```

- To overwrite the default level we have a method "uvm_top.set_report_verbosity_level(UVM_HIGH);"

"Code Practising":-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
  int verbosity;
  initial begin
    uvm_top.set_report_verbosity_level(UVM_HIGH);
    #10;
    `uvm_info("Chinnu","Hello",UVM_HIGH);#10;
  end
endmodule;
```

"Result":-

```
UVM_INFO /home/runner/testbench.sv(8) @ 10: reporter [Chinnu] Hello
Simulation has finished. There are no more test vectors to simulate.
```

- How we are going to send the values of the variables to the console.
- We have three different methods to do this
  - ✓ $sformat(Whenever we have single variable)
  - ✓ core method(Whenever we have to send transaction to the console)
  - ✓ do hooks("")

"Code Practising":-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
module tb;
  int data=0;
  initial begin
    `uvm_info("TB_TOP",$sformatf("The value of the data %0b",data),UVM_MEDIUM);
  end
endmodule
```

"Result":-

```
ASDB file was created in location /home/runner/dataset.asdb
UVM_INFO /home/runner/testbench.sv(6) @ 0: reporter [TB_TOP] The value of the data 00000000000b
Simulation has finished. There are no more test vectors to simulate.
```

“UVM: UVM Base Class”

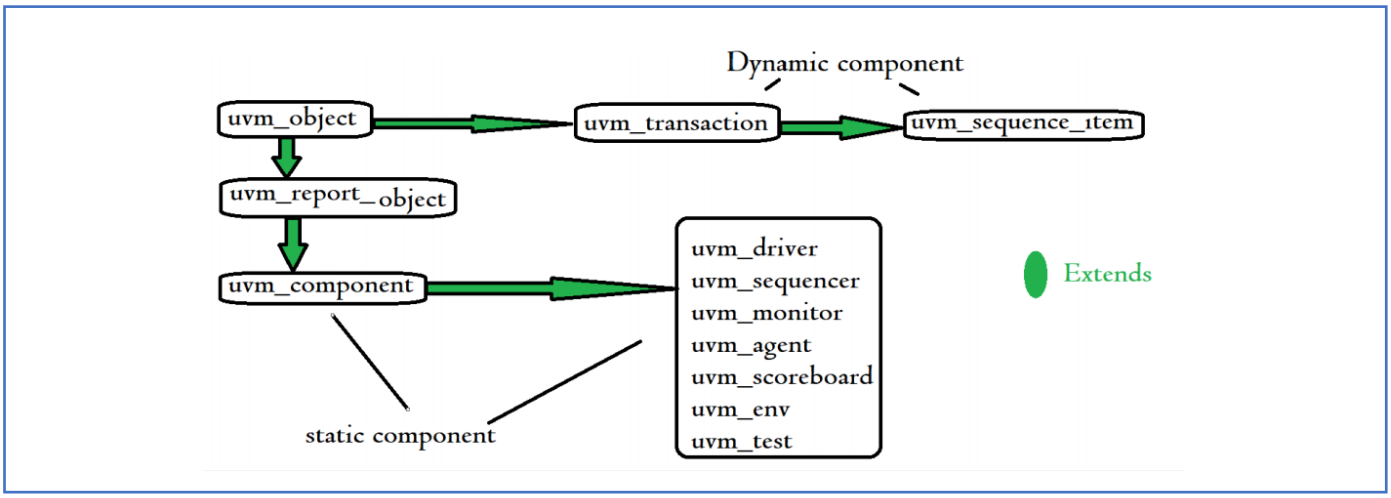
- Let’s discuss the importance of base class with the code

```
class add;

  bit [3:0] a;
  bit [3:0] b;
  bit [4:0] c;
  function new(bit [3:0] a, bit [3:0] b);
    this.a = a;
    this.b = b;
    this.c = a + b;
  endfunction
endclass
class mul extends add;
  bit [3:0] p;
  bit [3:0] q;
  bit [7:0] r;
  function new(bit [3:0] p, bit [3:0] q);
    super.new(p,q);
    this.r = p * q;
  endfunction
endclass
module tb;
  mul t;
  int a, m;
  initial begin
    t = new(4'b0001, 4'b0010);
    a = t.c;
    m = t.r;
    $display("add : %0d and mul : %0d",a,m);
  end
endmodule
```

- You can access the parent class properties using the child class.
- In UVM we represent the Generator as a sequence and Sequencer sends data from the sequence to the driver
- Base Classes: UVM Components & UVM Object:-
  - ➢ Static components:- Which stay in the verification environment till the process done is called static components
    - ✓ Eg:-Scoreboard, Driver, monitor, Sequencer
  - ➢ Dynamic components:-These varies in the verification environment
    - ✓ Eg:- Transaction
- Static components are built using UVM Components

- Dynamic components are built using UVM object
- Building the constructor is different for uvm_object and uvm_component
- Let's check the default constructor for the base classes
- For uvm_object:-
  - ✓ function new(input string name=”chinnu”);
    super.new(name)
- For uvm_component:-
  - ✓ function new(string name = “Chinnu”,uvm_component parent=null);
    super(name,parent)//This parent helps us in the uvm_tress



- **Building a class using uvm_object:-**

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class obj extends uvm_object;
  `uvm_object_utils(obj)
  function new(string inst="obj");
    super.new(inst);
  endfunction
endclass
module tb;
  obj o;
  initial begin
  o=new("O");
  end
endmodule
```

- **Building a class using uvm_component:-**

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class comp extends uvm_component;
  `uvm_component_utils(comp) // To register our class to factory
  function new(string inst="comp",uvm_component parent=null);
    super.new(inst,parent);
  endfunction
endclass
module tb;
  comp c;
  initial begin
    c=new("c",null);
  end
endmodule
```

“UVM: UVM Object(Part-1)”

- Core methods available in UVM:- Print, Copy, Compare, clone, Pack/Unpack, record, Create
- The space where we add field macros to our data members there we use utils
- The flags mentioned as arguments in a function, Check it

UVM_ALL_ON	Set all operations on.
UVM_DEFAULT	This is the recommended set of flags to pass to the field macros. Currently, it enables all of the operations, making it functionally identical to *UVM_ALL_ON*. In the future however, additional flags could be added with a recommended default value of *off*.
UVM_NOCOPY	Do not copy this field.
UVM_NOCOMPARE	Do not compare this field.
UVM_NOPRINT	Do not print this field.
UVM_NOPACK	Do not pack or unpack this field.
UVM_REFERENCE	For object types, operate only on the handle (e.g. no deep copy)
UVM_PHYSICAL	Treat as a physical field. Use physical setting in policy class for this field.
UVM_ABSTRACT	Treat as an abstract field. Use the abstract setting in the policy class for this field.
UVM_READONLY	Do not allow setting of this field from the set_*_local methods or during uvm_component::apply_config_settings operation.

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class transaction extends uvm_sequence_item;
  rand bit [3:0] data;
  function new (string inst="transaction");
    super.new(inst);
  endfunction
  `uvm_object_utils_begin(transaction)
  `uvm_field_int(data,UVM_DEFAULT)
  `uvm_object_utils_end
endclass
module tb;
  transaction t;
  initial begin
    t=new();
    t.randomize();
    t.print();
  end
endmodule
```

“Result”:-

```
----------------------------------
Name         Type         Size  Value
----------------------------------
transaction  transaction  -     @335
  data       integral     4     'h6
----------------------------------
Simulation has finished. There are no more test vectors to simulate.
```

- We can change the radix by using the following conventions

UVM_BIN	Print / record the field in binary (base-2).
UVM_DEC	Print / record the field in decimal (base-10).
UVM_UNSIGNED	Print / record the field in unsigned decimal (base-10)
UVM_OCT	Print / record the field in octal (base-8).
UVM_HEX	Print / record the field in hexadecimal (base-16).
UVM_STRING	Print / record the field in string format.
UVM_TIME	Print / record the field in time format.

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class transaction extends uvm_sequence_item;
  rand bit [3:0] data;
  function new (string inst="transaction");
    super.new(inst);
  endfunction
  `uvm_object_utils_begin(transaction)
  `uvm_field_int(data,UVM_DEFAULT|UVM_BIN) //**
  `uvm_object_utils_end
endclass
module tb;
  transaction t;
  initial begin
    t=new();
    t.randomize();
    t.print();
  end
endmodule
```

“Result”:-

```
----------------------------------
Name         Type         Size  Value
----------------------------------
transaction  transaction  -     @335
  data       integral     4     'b110
----------------------------------
```

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class transaction extends uvm_sequence_item;
  rand bit [3:0] data;
  function new (string inst="transaction");
    super.new(inst);
  endfunction
  `uvm_object_utils_begin(transaction)
  `uvm_field_int(data,UVM_DEFAULT|UVM_BIN) //**
  `uvm_object_utils_end
endclass
module tb;
  transaction t;
  initial begin
    t=new();
    t.randomize();
    t.print(uvm_default_tree_printer);//Tree Printer
  end
endmodule
```

“Result”:-

```
transaction: (transaction@335) {
  data: 'b110
}
Simulation has finished. There are no more test vectors to simulate.
```

- Inbuilt implementations are less efficient than compared to do

“Do Hooks Print”:-

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class transaction extends uvm_sequence_item;
  rand bit [3:0] data;
  function new (string inst="transaction");
    super.new(inst);
  endfunction
  `uvm_object_utils_begin(transaction)
  `uvm_field_int(data,UVM_DEFAULT|UVM_BIN) //**
  `uvm_object_utils_end
  virtual function void do_print(uvm_printer printer);
    super.do_print(printer);
    printer.print_field("data",data,$bits(data),UVM_DEC);
  endfunction
endclass
module tb;
  transaction t;
  initial begin
    t=new("t");
    t.randomize();
    t.print(uvm_default_tree_printer);//Tree Printer
  end
endmodule
```

“Result”:-

```
t: (transaction@335) {
  data: 'b110
  data: 'd6
}
```

“UVM: UVM Object(Part-2)”

“Inbuild Implementation: Copy”:-

Shallow vs Deep copy:-

- In the case of Deep Copy we have independent instances of the class
- In a Shallow copy we have single instances that are accessible from all the instances that copy the original class.

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class temp extends uvm_object;
  bit [3:0] data_temp;
  function new(string inst = "temp");
  super.new(inst);
  endfunction
  `uvm_object_utils_begin(temp)
  `uvm_field_int(data_temp, UVM_DEFAULT)
  `uvm_object_utils_end
endclass
class transaction extends uvm_sequence_item;
 rand bit [3:0] data;
 temp t;
function new(string inst = "transaction");
super.new(inst);
t = new("t");
endfunction
  `uvm_object_utils_begin(transaction)
  `uvm_field_int(data, UVM_DEFAULT)
  `uvm_field_object(t, UVM_DEFAULT)
  `uvm_object_utils_end
endclass
module tb;
transaction tr_a, tr_b;
initial begin
  ////////adding constructor to both instances
  tr_a = new("tr_a");
  tr_b = new("tr_b");
  ///////////////generate random data for one of instance
  tr_a.randomize();
  tr_a.t.data_temp = 4'b0011;
  tr_a.print();
  ////////////////////copy the content of instance a to b
  tr_b.copy(tr_a);
  tr_b.print();
  ////////////////////update the content from any one instance
```

“Result”:-

```
----------------------------------------
Name           Type         Size  Value
----------------------------------------
tr_a           transaction  -     @335
  data         integral     4     'h6
  t            temp         -     @338
    data_temp  integral     4     'h3
----------------------------------------
----------------------------------------
Name           Type         Size  Value
----------------------------------------
tr_b           transaction  -     @339
  data         integral     4     'h6
  t            temp         -     @341
    data_temp  integral     4     'h3
----------------------------------------
----------------------------------------
Name           Type         Size  Value
----------------------------------------
tr_a           transaction  -     @335
  data         integral     4     'h6
  t            temp         -     @338
    data_temp  integral     4     'h3
----------------------------------------
----------------------------------------
Name           Type         Size  Value
----------------------------------------
tr_b           transaction  -     @339
  data         integral     4     'h6
  t            temp         -     @341
    data_temp  integral     4     'h0
----------------------------------------
Simulation has finished. There are no more test vectors to simulate.
```

“Do Hook: Copy”:-

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class temp extends uvm_object;
 `uvm_object_utils(temp)
  bit [3:0] data_temp;
  function new(string inst = "temp");
  super.new(inst);
  endfunction
  virtual function void do_print(uvm_printer printer);
  super.do_print(printer);
  printer.print_field("data_temp", data_temp, $bits(data_temp), UVM_DEC);
  endfunction
endclass
class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)
 rand bit [3:0] data;
 temp t;
function new(string inst = "transaction");
super.new(inst);
t = new("t");
endfunction
  virtual function void do_print(uvm_printer printer);
  super.do_print(printer);
  printer.print_field("data", data, $bits(data), UVM_DEC);
  printer.print_object("t", t);
  endfunction
  virtual function void do_copy(uvm_object rhs);
    transaction tr;
    super.do_copy(rhs);
    $cast(tr, rhs);
    data = tr.data;
    t.data_temp = tr.t.data_temp;
  endfunction
endclass
module tb;
transaction tr_a, tr_b;
initial begin
  ////////adding constructor to both instances
  tr_a = new("tr_a");
  tr_b = new("tr_b");
  ///////////////generate random data for one of instance
  tr_a.randomize();
  tr_a.t.data_temp = 4'b0011;
  tr_a.print();
  /////////////////copy the content of instance a to b
  tr_b.copy(tr_a);
  tr_b.print();
  /////////////////update the content from any one instance
  tr_b.t.data_temp = 4'b0000;
  tr_a.print();
  tr_b.print();
end
endmodule
```

“Result”:-

```
Name            Type         Size  Value
----------------------------------------
tr_a            transaction  -     @335
  data          integral     4     'h6
  t             temp         -     @338
    data_temp   integral     4     'h3
----------------------------------------
----------------------------------------
Name            Type         Size  Value
----------------------------------------
tr_b            transaction  -     @339
  data          integral     4     'h6
  t             temp         -     @341
    data_temp   integral     4     'h3
----------------------------------------
----------------------------------------
Name            Type         Size  Value
----------------------------------------
tr_a            transaction  -     @335
  data          integral     4     'h6
  t             temp         -     @338
    data_temp   integral     4     'h3
----------------------------------------
----------------------------------------
Name            Type         Size  Value
----------------------------------------
tr_b            transaction  -     @339
  data          integral     4     'h6
  t             temp         -     @341
    data_temp   integral     4     'h0
```

“Create Method”:-

“Code Practising”:-

```
`include "uvm_macros.svh"
 import uvm_pkg::*;
////////////////////////////////////////////
////all the classes -> create
////tlm ports -> new
class transaction extends uvm_sequence_item;
  rand bit [3:0] a;
  rand bit [3:0] b;
       bit [4:0] y;
  function new(input string inst = "transaction");
  super.new(inst);
  endfunction
`uvm_object_utils_begin(transaction)
  `uvm_field_int(a,UVM_DEFAULT)
  `uvm_field_int(b,UVM_DEFAULT)
  `uvm_field_int(y,UVM_DEFAULT)
`uvm_object_utils_end
endclass
module tb;
  transaction tr;
  initial begin
    tr = transaction::type_id::create("tr");
    tr.randomize();
    tr.print();
  end
endmodule
```

“Result”:-

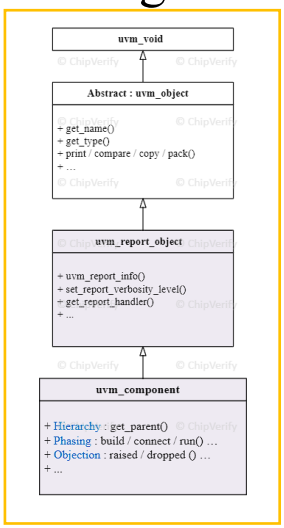
```
----------------------------
Name  Type         Size  Value
----------------------------
tr    transaction  -     @335
  a   integral     4     'h6
  b   integral     4     'h5
  y   integral     5     'h0
----------------------------
Simulation has finished. There are no more test vectors to simulate.
```

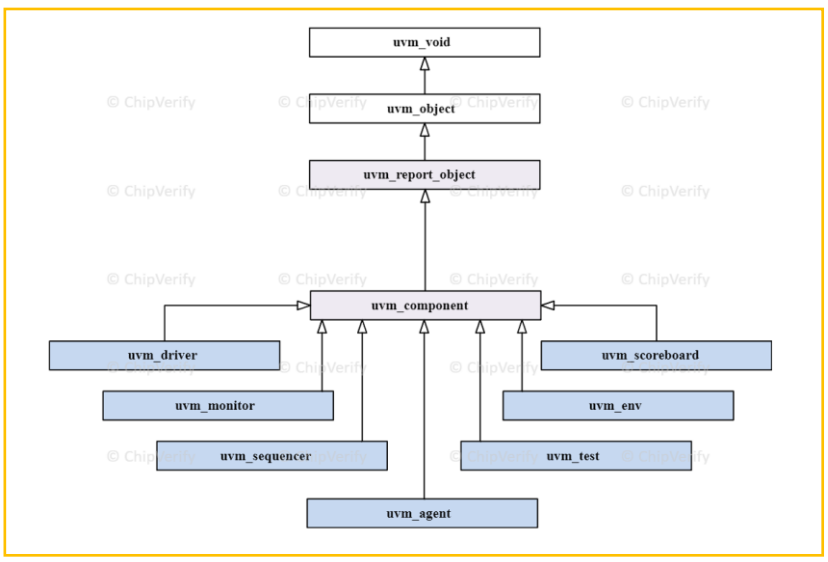
“UVM: UVM Component(Part-1)”

uvm_component is a fundamental base class that serves as the foundation for all UVM components like drivers, monitors and scoreboards in a verification environment. It has the following features:

- **Hierarchy:-**

  Supports a hierarchical structure, where each component can have child components, forming a tree-like structure and provides methods for searching and traversing the tree.





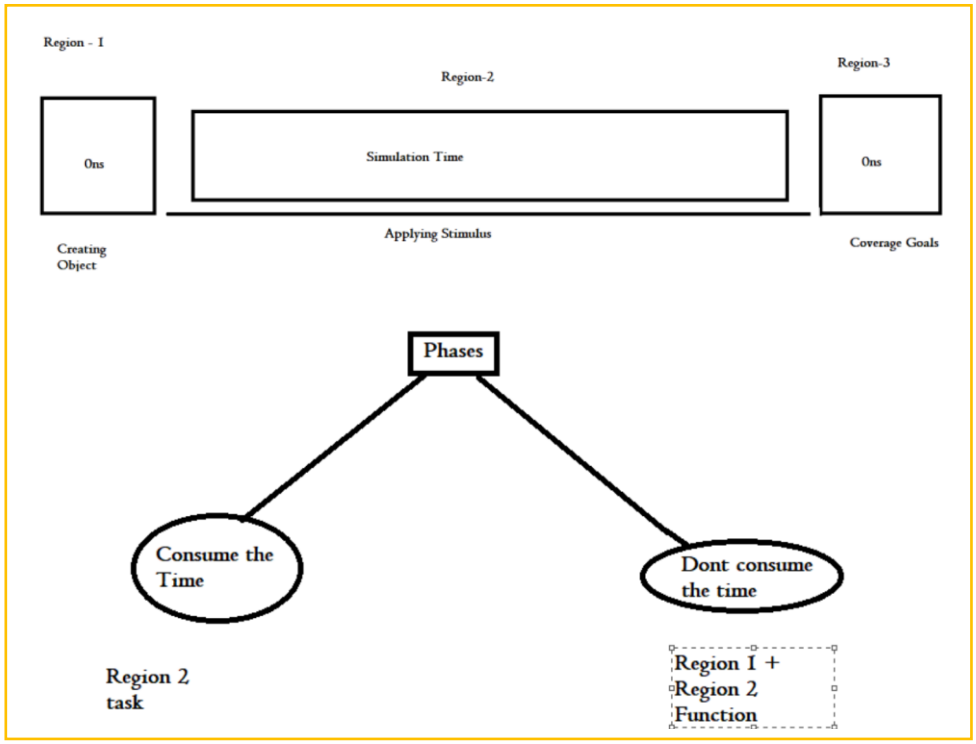
- **Phasing:**
  Components can participate in the UVM phasing mechanism, which organizes the simulation into different phases like build, connect, run, and cleanup. Components can perform specific tasks during each phase.
- **Reporting:-**
  Components can use the UVM messaging infrastructure to report events, warnings, and errors during simulation.
- **Factory:-**
  Components can be registered with the UVM factory mechanism, enabling dynamic object creation and lookup

"Code Practising":-

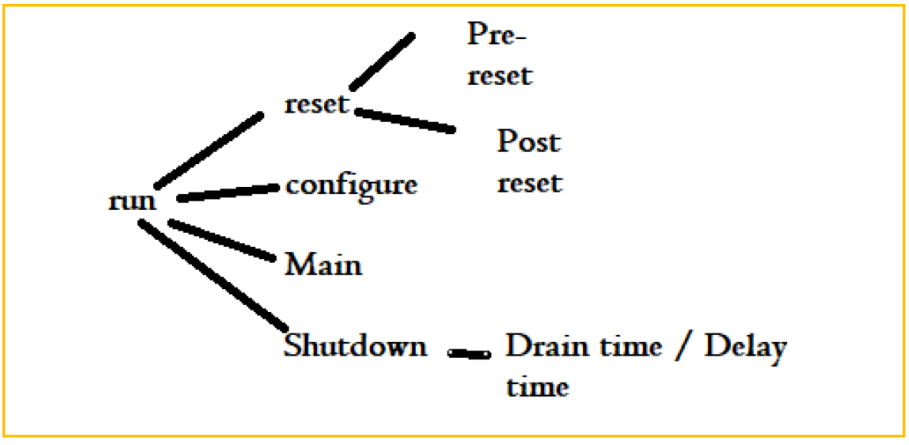
```
// File: my_component.sv
// Include UVM library
`include "uvm_macros.svh"
`include "uvm_pkg.sv"
// Define your component as a UVM agent
class my_component extends uvm_agent;
  // Declare your component-specific variables and components here
  // For example, you may have interface handles, monitors, drivers, etc.
  // Define the constructor for your component
  function new(string name = "my_component", uvm_component parent = null);
    super.new(name, parent);
  endfunction
  // Define the build_phase to configure and create sub-components
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Create and configure your sub-components (e.g., driver, monitor, sequencer)
    // Example:
    // my_driver driver_inst;
    // my_monitor monitor_inst;
    // my_sequencer sequencer_inst;
    // Set the connections between components if needed
  endfunction
  // Define the connect_phase to establish connections between components
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    // Connect the components, if not done in build_phase
  endfunction
  // Define the run_phase for the main functionality of your component
  virtual task run_phase(uvm_phase phase);
    // Implement the main functionality of your component here
    // This could include generating stimulus, driving the DUT, monitoring signals, etc.
    // Example:
    // my_sequencer.start(env_config.sequencer_cfg);
    // my_sequencer.wait_for_sequences();
  endtask
  // Optionally, you can implement other UVM phases and tasks as needed
endclass
```

“UVM: UVM Component(Part-2)”

- UVMPhase+UVMTree are the extra added features for the uvm component.



Region-1:-build, connect, End of elaboration, Start of simulation
Region-2:-Extract, check, report, Final
These regions need function as they don’t need any time
Region-2:-



All these are executed in sequential order in not parallel

- Build phase => Create an object of class
- Connect Phase => Connection of port
- End of Elaboration => Analyze the hierarchy
- Start of simulation => to initialize the variable or to create the memory
- Run => reset -> apply stimulus -> Collect responses
- Cleanup Phases => General Report / Coverage Goal

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class test extends uvm_test;
  `uvm_component_utils(test)
  function new(string path = "test", uvm_component parent = null);
    super.new(path, parent);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("test","Build Phase Executed", UVM_NONE);
  endfunction
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("test","Connect Phase Executed", UVM_NONE);
  endfunction
  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    `uvm_info("test","End of Elaboration Phase Executed", UVM_NONE);
  endfunction
   virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    `uvm_info("test","Start of Simulation Phase Executed", UVM_NONE);
  endfunction
  virtual task run_phase(uvm_phase phase);
    `uvm_info("test", "Run Phase Executed", UVM_NONE);
  endtask
 virtual function void extract_phase(uvm_phase phase);
    super.extract_phase(phase);
    `uvm_info("test", "Extract Phase", UVM_NONE);
  endfunction
 virtual function void check_phase(uvm_phase phase);
    super.check_phase(phase);
    `uvm_info("test", "Check Phase", UVM_NONE);
  endfunction
 virtual function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    `uvm_info("test", "Report Phase", UVM_NONE);
  endfunction
  virtual function void final_phase(uvm_phase phase);
    super.final_phase(phase);
      `uvm_info("test", "Final Phase", UVM_NONE);
  endfunction
endclass
module tb;
  initial begin
    run_test("test");
  end
endmodule
```

“Result”:-

```
UVM_INFO @ 0: reporter [RNTST] Running test test...
UVM_INFO /home/runner/testbench.sv(10) @ 0: uvm_test_top [test] Build Phase Executed
UVM_INFO /home/runner/testbench.sv(14) @ 0: uvm_test_top [test] Connect Phase Executed
UVM_INFO /home/runner/testbench.sv(18) @ 0: uvm_test_top [test] End of Elaboration Phase Executed
UVM_INFO /home/runner/testbench.sv(22) @ 0: uvm_test_top [test] Start of Simulation Phase Executed
UVM_INFO /home/runner/testbench.sv(25) @ 0: uvm_test_top [test] Run Phase Executed
UVM_INFO /home/runner/testbench.sv(29) @ 0: uvm_test_top [test] Extract Phase
UVM_INFO /home/runner/testbench.sv(33) @ 0: uvm_test_top [test] Check Phase
UVM_INFO /home/runner/testbench.sv(37) @ 0: uvm_test_top [test] Report Phase
UVM_INFO /home/runner/testbench.sv(41) @ 0: uvm_test_top [test] Final Phase
UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_report_server.svh(869) @ 0: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   11
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[UVM/RELNOTES]     1
[test]     9
```

- **Simulating for more than 0 n sec:-**

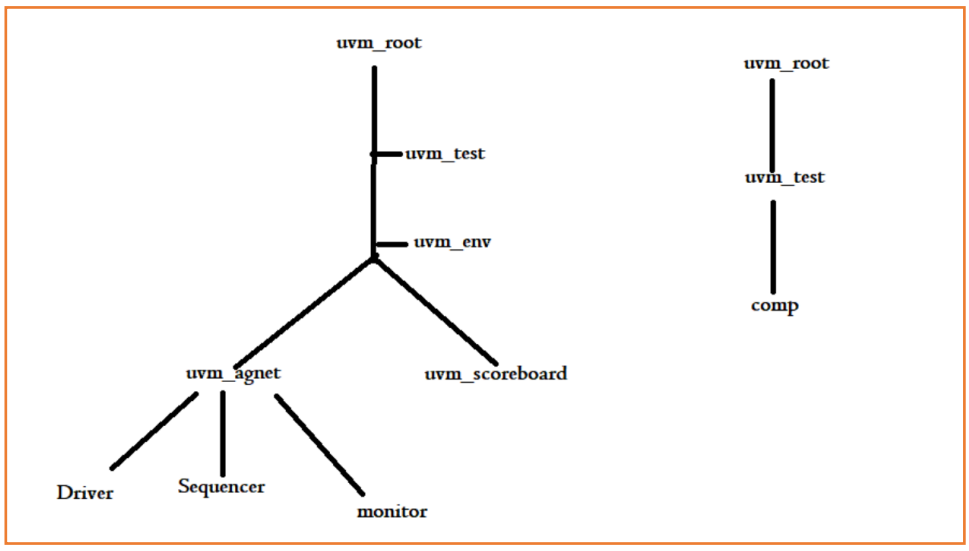
“Code Practising”:-

```
////////////////////////////////////////////////////
`include "uvm_macros.svh"
import uvm_pkg::*;
/////////////////////////////////////////////////////////////////////////
class test extends uvm_test;
`uvm_component_utils(test)
  function new(input string inst = "test", uvm_component parent = null);
    super.new(inst,parent);
  endfunction
  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
     #10;
    `uvm_info("TEST", "Executed Test Run Phase", UVM_NONE);
    phase.drop_objection(this);
  endtask
endclass
////////////////////////////////////////////////
module tb;
initial begin
run_test("test");
end
endmodule
```

“Result”:-

```
ASDB file was created in location /home/runner/dataset.asdb
UVM_INFO @ 0: reporter [RNTST] Running test test...
UVM_INFO /home/runner/testbench.sv(13) @ 10: uvm_test_top [TEST] Executed Test Run Phase
UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_objection.svh(1271) @ 10: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_report_server.svh(869) @ 10: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    4
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST]     1
[TEST_DONE]     1
[UVM/RELNOTES]     1
```



- By defining only one root we can achieve a single standard verification methodology.
- We can’t directly take uvm_root, so UVM has a universal class called uvm_top. Using this we can make uvm_root a parent class

“Code Practising”:-

```
`include "uvm_macros.svh"
import uvm_pkg::*;
class comp extends uvm_component;
  `uvm_component_utils(comp)
  function new(string inst = "comp", uvm_component parent = null);
    super.new(inst, parent);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("COMP", "COMP BUILD PHASE EXECUTED", UVM_NONE);
  endfunction
endclass
class test extends uvm_test;
  `uvm_component_utils(test)
  comp c;
  function new(string inst = "test", uvm_component parent = null);
    super.new(inst, parent);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    c = comp::type_id::create("c", this);
    `uvm_info("TEST", "TEST BUILD PHASE EXECUTED", UVM_NONE);
  endfunction

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    uvm_top.print_topology();
  endfunction
endclass
module tb;
  initial begin
    run_test("test");
  end
endmodule
```

“Reference”:-

https://www.udemy.com/share/105G4g3@GVFxAk7r9EDrA0C0mmIhykUyDhagH9KVJF9y4xqUa1kIjvAJlngVHo6o3_b-KzeWxA==/

"UVM: UVM Resource Sharing"

"Resource Sharing":-

- Interface is shared with config_db
- Transaction is shared along the classes using TLM_Ports
- Mailbox is like TLM_ports
- For Dynamic things we need to add the virtual keyword for it.

"Config_db":-

- It has two methods called set and gets
- Set is used in the testbench top to retrieve the data
- Get is used in the monitor to retrieve the data
- Let's see the arguments used in the set method and get the method

```
Set(context,inst name,Key,Value)
```

- If {context + inst_name + Key} doesn't match for the get and set method, Then it gives a fatal error
- Context has two keywords they are "null and this "

Eg:-

```
set(null,"uvm_test_top.env.agent.mon.a","key",__);
get(this,"a","key",__)
```

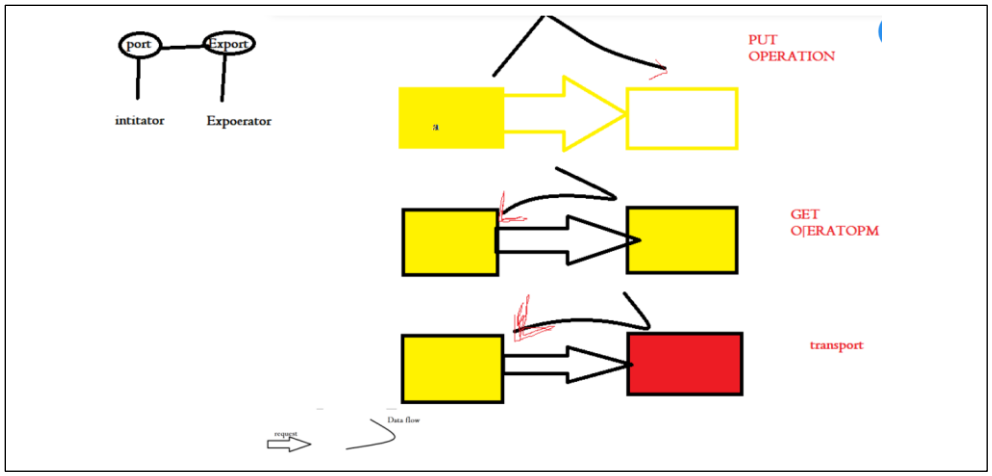
"Code Practising":-

```
module adder (
  input [3:0] a,b,
  output [4:0]y
);
  assign y=a+b;
endmodule
interface adder_if;
  logic [3:0] a;
  logic [3:0] b;
  logic [4:0] y;
endinterface
```

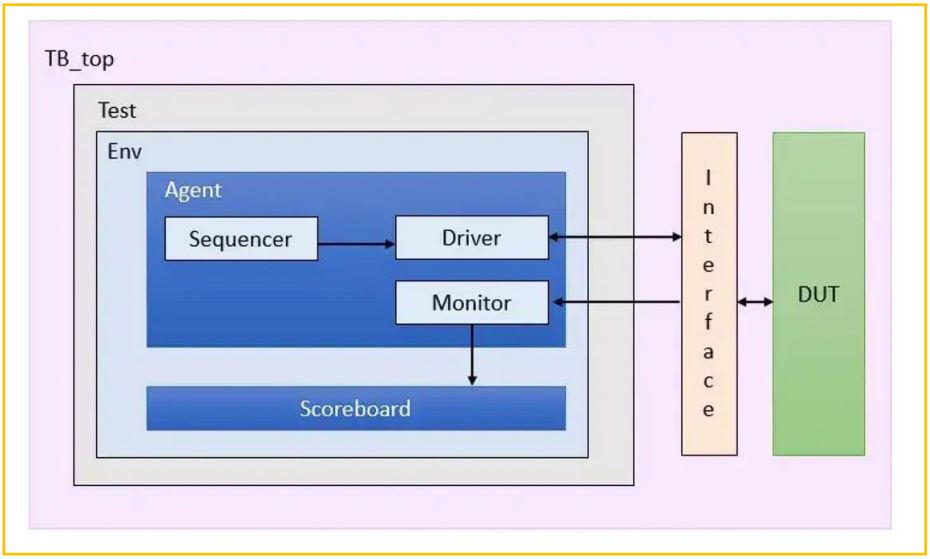
```
`include "uvm_macros.svh"
import uvm_pkg::*;
class drv extends uvm_driver;
  `uvm_component_utils(drv)
  virtual adder_if aif;
  function new(input string path = "drv", uvm_component parent = null);
    super.new(path,parent);
  endfunction
  virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
    if(!uvm_config_db#(virtual adder_if)::get(this,"","aif",aif))//drv.aif
      `uvm_error("drv","Unable to access Interface");
  endfunction
   virtual task run_phase(uvm_phase phase);
     phase.raise_objection(this);
     for(int i = 0; i< 10; i++)
      begin
        aif.a <= $urandom;
        aif.b <= $urandom;
        #10;
      end
     phase.drop_objection(this);
   endtask
endclass
//////////////////////////////////////////////////////////////////////
module tb;
  adder_if aif();
  adder dut (.a(aif.a), .b(aif.b), .y(aif.y));
initial
  begin
    uvm_config_db #(virtual adder_if)::set(null, "uvm_test_top", "aif", aif);//uvm_test_top.aif
    run_test("drv");
  end
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
  end
endmodule
```

“Fundamentals in TLM Ports”:-

- TLM is commonly used in communication between the scoreboard and monitor



“UVM: UVM Testbench Components”



- **Transaction:-**
  Keep Track of all the I/O present in DUT
- **Sequence:**
  Combination of transactions to verify the specific test case. Transaction and Sequence are the dynamic components Sequence overflow error will be thrown if you keep trans as an argument in an item done by the driver.
- **Sequencer:-**
  Manage sequences and send the sequence to the driver after request (uvm_sequencer)
- **Driver:-**
  Send request to driver for sequence, apply the sequence to the DUT (uvm_driver)
- **Monitor:-**
  Collect the response of DUT and forward it to the scoreboard (uvm_monitor).
- **Scoreboard:-**
  Compare response with golden data(uvm_scoreboard).

THANK YOU !

The VLSI Voyager

chinna venkata narayana reddy seelam
cvnreddyseelam07@gmail.com