Q1: What is UVM? What is the advantage of UVM?

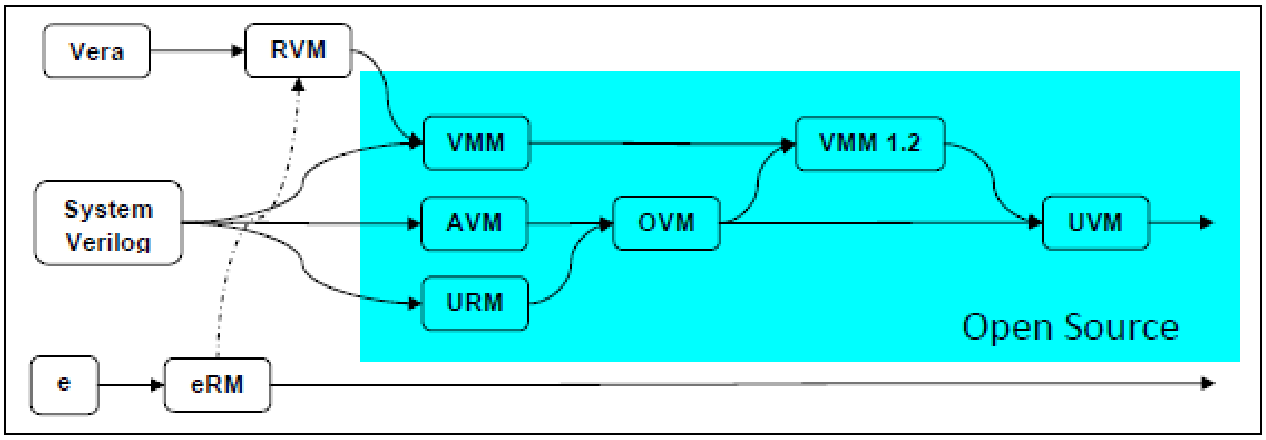
Ans: UVM (Universal Verification Methodology) is a standardized methodology for verifying the both complex & simple digital design in simple way.

UVM Features:

- First methodology & second collection of class libraries for Automation
- Reusability through testbench
- Plug & Play of verification IPs
- Generic Testbench Development
- Vendor & Simulator Independent
- Smart Testbench i.e. generate legal stimulus as from pre-planned coverage plan
- Support CDV –Coverage Driven Verification
- Support CRV –Constraint Random Verification
- UVM standardized under the Accellera System Initiative
- Register modeling

Q2: UVM derived from which language?

Ans: Here is the detailed connection between SV, UVM, OVM and other methodologies.



Q3. What is the difference between uvm_component and uvm_object?
OR
We already have uvm_object, why do we need uvm_component which is actually derived class of uvm_object?

Ans:
uvm_component:

- Quasi Static Entity (after build phase it is available throughout the simulation)
- Always tied to a given hardware(DUT Interface) Or a TLM port

- Having phasing mechanism for control the behavior of simulation
- Configuration Component Topology

uvm_object:

- Dynamic Entity (create when needed, transfer from one component to other & then dereference)
- Not tied to a given hardware or any TLM port
- Not phasing mechanism

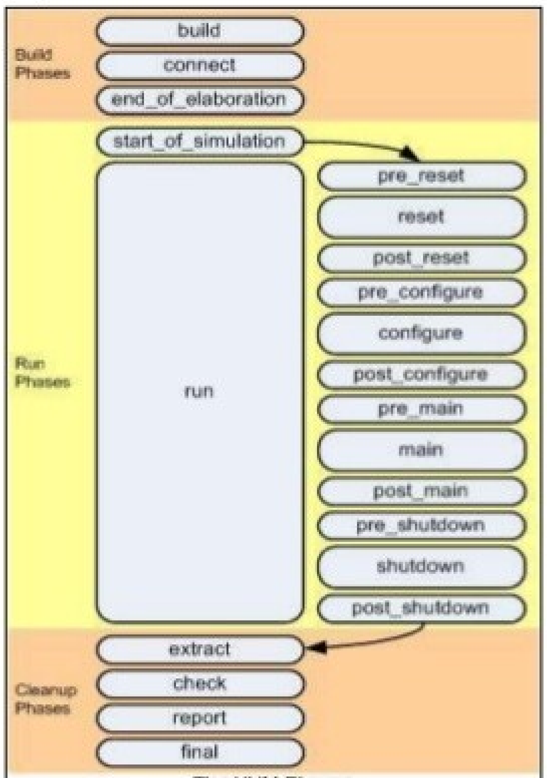
Q4: Why phasing is used? What are the different phases in uvm?

Ans: UVM Phases is used to control the behavior of simulation in a systematic way & execute in a sequential ordered to avoid race condition. This could also be done in system verilog but manually.

1. List of UVM Phases:
2. buid_phase
3. connect_phase
4. end_of_elaboration_phase
5. start_of_simulation_phase
6. run _phase (task)
   Sub Phases of Reset Phase:
   pre_reset_phase
   reset_phase
   post_reset_phase
   pre_configure_phase
   configure_phase
   post_configure_phase
   pre_main_phase
   main_phase
   post_main_phase
   pre_shutdown_phase
   shutdown_phase
   post_shutdown_phase
7. extract_phase
8. check_phase
9. report_phase

Below figure makes it more clear

UVM Phases



The UVM Phases

- Build Phases
  - Build Top-Level Testbench Topology
  - Connect environment topology
  - Post-elaboration activity (e.g. print topology)
- Run Phases
  - Run-time execution of test
  - All phases except run() execute in zero time
  - Lots of sub-phases not used really
- Cleanup Phases
  - Gathers details on the final DUT state
  - Processes and checks the simulation results.
  - Simulation results analysis and reporting

Q5: Which uvm phase is top - down , bottom – up & parallel?

Ans: Only build phase is a top-down & other phases are bottom-up except run phase which is parallel. The build phase works top-down since the testbench hierarchy may be configure so we need to build the branches before leafs

Q6: Why build phase is top – down & connect phase is bottom – up?

Ans: The connect phase is intended to be used for making TLM connections between components, which is why it occur after build phase. It work bottom-up so that its got the correct implementation all the way up the design hierarchy, if worked top-down this would be not possible

Q7: Which phase is function & which phase is task?

Ans: Only run phase is a task (time consuming phase) & other phases are functions (non-blocking)

Q8: Which phase takes more time and why?

Ans: As previously said the run phase is implemented as task and remaining all are function. run phase will get executed from start of simulation to till the end of simulation. run phase is time consuming, where the testcase is running.

Q9: How uvm phases initiate?

Ans: UVM phases initiate by calling run_test(“test1”) in top module. When run_test() method call, it first create the object of test top & then call all phases.

Q10: How test cases run from simulation command line?

Ans: In top module write run_test(); i.e. Don't give anything in argument.
Then in command line : +UVM_TESTNAME=testname

Q11: Difference between module & class based TB?

Ans: A module is a static object present always during of the simulation.
A Class is a dynamic object because they can come and go during the life time of simulation.

Q12: What is uvm_config_db ? What is difference between uvm_config_db & uvm_resource_db?

Ans: uvm_config_db is a parameterized class used for configuration of different type of parameter into the uvm database, So that it can be used by any component in the lower level of hierarchy.

uvm_config_db is a convenience layer built on top of uvm_resource_db, but that convenience is very important. In particular, uvm_resource_db uses a "last write wins" approach. The uvm_config_db, on the other hand, looks at where things are in the hierarchy up through end_of_elaboration, so "parent wins." Once you start start_of_simulation, the config_db becomes "last write wins."

All of the functions in uvm_config_db#(T) are static, so they must be called using the :: operator
It is extended from the uvm_resource_db#(T), so it is child class of uvm_resource_db#(T)

Q13: What is the advantage and difference of `uvm_component_utils() and `uvm_object_utils()?

Ans: The utils macros define the infrastructure needed to enable the object/component for correct factory operation.

The reason there are two macros is because the factory design pattern fixes the number of arguments that a constructor can have. Classes derived from uvm_object have constructors with one argument, a string name. Classes derived from uvm_component have two arguments, a name and a uvm_component parent.

The two `uvm_*utils macros inserts code that gives you a factory create() method that delegates calls to the constructors of uvm_object or uvm_component. You need to use the respective macro so that the correct constructor arguments get passed through. This means that you cannot add extra constructor arguments when you extend these classes in order to be able to use the UVM factory.

Q14: Difference between `uvm_do and `uvm_rand_send ?

Ans: `uvm_do perform the below steps:

1. create
2. start_item
3. randomize
4. finish_item
5. get_response (optional)

while `uvm_rand_send perform all the above steps except create. User needs to create sequence / sequence_item.

Q15: Difference between uvm_transaction and uvm_seq_item?

Ans: *class uvm_sequence_item extends uvm_transaction*

uvm_sequence_item extended from uvm_transaction only, uvm_sequence_item class has more functionality to support sequence & sequencer features. uvm_sequence_item provides the hooks for sequencer and sequence , So you can generate transaction by using sequence and sequencer , and uvm_transaction provide only basic methods like do_print and do_record etc .

Q16: Is uvm is independent of systemverilog ?

Ans: UVM is a methodology based on SystemVerilog language and is not a language on its own. It is a standardized methodology that defines several best practices in verification to enable efficiency in terms of reuse and is also currently part of IEEE 1800.2 working group.

Q17: What are the benefits of using UVM?

Ans: Some of the benefits of using UVM are :

- **Modularity and Reusability –** The methodology is designed as modular components (Driver, Sequencer, Agents , env etc) which enables reusing components across unit level to multi-unit or chip level verification as well as across projects.
- **Separating Tests from Testbenches –** Tests in terms of stimulus/sequencers are kept separate from the actual testbench hierarchy and hence there can be reuse of stimulus across different units or across projects
- **Simulator independent –** The base class library and the methodology is supported by all simulators and hence there is no dependence on any specific simulator
- **Better control on Stimulus generation –** Sequence methodology gives good control on stimulus generation. There are several ways in which sequences can be developed which includes randomization, layered sequences, virtual sequences etc which provides a good control and rich stimulus generation capability.
- **Easy configuration –** Config mechanisms simplify configuration of objects with deep hierarchy. The configuration mechanism helps in easily configuring different testbench components based on which verification environment uses it and without worrying about

how deep any component is in testbench hierarchy

- **Factory mechanism –** Factory mechanisms simplifies modification of components easily. Creating each components using factory enables them to be overridden in different tests or environments without changing underlying code base.

Q18: Can we have user defined phase in UVM?

Ans: In addition to the predefined phases available in uvm , the user has the option to add his own phase to a component. This is typically done by extending the uvm_phase class the constructor needs to call super.new which has three arguments

- Name of the phase task or function
- Top down or bottom up phase
- Task or function

The call_task or call_func and get_type_name need to be implemented to complete the addition of new phase.

Example

```
class custom_phase extends uvm_phase;
  function new();
    super.new("custom",1,1);
  endfunction

  task call_task  ( uvm_component parent);
    my_comp_type comp;
    if ( $cast(comp,parent) )
          comp.custom_phase();
  endtask

  virtual function string get_type_name();
    return "custom";
  endfunction
endclass
```

Q19: What is uvm RAL model ? why it is required ?

Ans: In a verification context, a register model (or register abstraction layer) is a set of classes that model the memory mapped behavior of registers and memories in the DUT in order to facilitate stimulus generation and functional checking (and optionally some aspects of functional coverage).

The UVM provides a set of base classes that can be extended to implement comprehensive register modeling capabilities.

Q20: What is the difference between new() and create?

Ans: We all know about new() method that is use to allocate memory to an object instance. In UVM (and OVM), the create() method causes an object instance to be created from the factory. This allows you to use factory overrides to replace the desired object with an object of a different type without having to recode.

Q21: What is analysis port?

Analysis port (class uvm_tlm_analysis_port) — a specific type of transaction-level port that can be connected to zero, one, or many analysis exports and through which a component may call the method write implemented in another component, specifically a subscriber.

port, export, and imp classes used for transaction analysis.

uvm_analysis_port
Broadcasts a value to all subscribers implementing a uvm_analysis_imp.
uvm_analysis_imp
Receives all transactions broadcasted by a uvm_analysis_port.
uvm_analysis_export
Exports a lower-level uvm_analysis_imp to its parent.

Q22: What is TLM FIFO?

In simpler words TLM FIFO is a FIFO between two UVM components, preferably between Monitor and Scoreboard. Monitor keep on sending the DATA, which will be stored in TLM FIFO, and Scoreboard can get data from TLM FIFO whenever needed.

```
// Create a FIFO with depth 4
    tlm_fifo = new ("uvm_tlm_fifo", this, 4);
```

Q23: How sequence starts?
start_item starts the sequence

```
virtual task start_item ( uvm_sequence_item item,
                            int  set_priority =  -1,
                           uvm_sequencer_base  sequencer =  null )
```

start_item and finish_item together will initiate operation of a sequence item. If the item has not already been initialized using create_item, then it will be initialized here to use the default sequencer specified by m_sequencer.

Q24: What is the difference between UVM RAL model backdoor write/read and front door write/read ?

Fontdoor access means using the standard access mechanism external to the DUT to read or write to a register. This usually involves sequences of time-consuming transactions on a bus interface.

Backdoor access means accessing a register directly via hierarchical reference or outside the language via the PLI. A backdoor reference usually in 0 simulation time.

Q25: What is objection?

The objection mechanism in UVM is to allow hierarchical status communication among components which is helpful in deciding the end of test.

There is a built-in objection for each in-built phase, which provides a way for components and objects to synchronize their testing activity and indicate when it is safe to end the phase and, ultimately, the test end.

The component or sequence will raise a phase objection at the beginning of an activity that must be completed before the phase stops, so the objection will be dropped at the end of that activity. Once all of the raised objections are dropped, the phase terminates.

Raising an objection: phase.raise_objection(this);
Dropping an objection: phase.drop_objection(this);

Q26: What is p_sequencer ? OR Difference between m_sequencer and p_sequencer?

m_sequencer is the default handle for uvm_vitual_sequencer and p_sequencer is the hook up for child sequencer.

m_sequencer is the generic uvm_sequencer pointer. It will always exist for the uvm_sequence and is initialized when the sequence is started.

p_sequencer is a typed-specific sequencer pointer, created by registering the sequence to the sequencer using macros (`uvm_declare_p_sequencer) . Being type specific, you will be able to access anything added to the sequencer (i.e. pointers to other sequencers, etc.). p_sequencer will not exist if we have not registered the sequence with the `uvm_declare_p_sequencer macros.

The drawback of p_sequencer is that once the p_sequencer is defined, one cannot run the sequence on any other sequencer type.

Q27: What is the difference between Active mode and Passive mode with respect to agent?

An agent is a collection of a sequencer, a driver and a monitor.

In **active mode**, the sequencer and the driver are constructed and stimulus is generated by sequences sending sequence items to the driver through the sequencer. At the same time the monitor assembles pin level activity into analysis transactions.

In **passive mode**, only the monitor is constructed and it performs the same function as in an active agent. Therefore, your passive agent has no need for a sequencer. You can set up the monitor using a configuration object.

Q28: What is the difference between copy and clone?

The built-in **copy()** method executes the __m_uvm_field_automation() method with the required copy code as defined by the field macros (if used) and then calls the built-in **do_copy()** virtual function. The built-in do_copy() virtual function, as defined in the uvm_object base class, is also an empty method, so if field macros are used to define the fields of the transaction, the built-in copy() method will be populated with the proper code to copy the transaction fields from the field macro definitions and then it will execute the empty do_copy() method, which will perform no additional activity.

The **copy()** method can be used as needed in the UVM testbench. One common place where the copy() method is used is to copy the sampled transaction and pass it into a sb_calc_exp() (scoreboard calculate expected) external function that is frequently used by the scoreboard predictor.

The **clone()** method calls the create() method (constructs an object of the same type) and then calls the copy() method. It is a one-step command to create and copy an existing object to a new object handle.

Q29: What is UVM factory?

UCM Factory is used to manufacture (create) UVM objects and components. Apart from creating the UVM objects and components the factory concept essentially means that you can modify or substitute the nature of the components created by the factory without making changes to the testbench.

For example, if you have written two driver classes, and the environment uses only one of them. By registering both the drivers with the factory, you can ask the factory to substitute the existing driver in environment with the other type. The code needed to achieve this is minimal, and can be written in the test.

Q30: What are the types of sequencer? Explain each?

There are two types of sequencers :

uvm_sequencer #(REQ, RSP) :
When the driver initiates new requests for sequences, the sequencer selects a sequence from a list of available sequences to produce and deliver the next item to execute. In order to do this, this type of sequencer is usually connected to a driver uvm_driver #(REQ, RSP).

uvm_push_sequencer #(REQ, RSP) :
The sequencer pushes new sequence items to the driver, but the driver has the ability to block the item flow when its not ready to accept any new transactions. This type of sequencer is connected to a driver of type uvm_push_driver #(REQ, RSP).

SV

What Is Callback ?

In computer programming, a callback is executable code that is passed as an argument to other code. It allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer.

What Is Factory Pattern ?

Factory Pattern Concept :

Methodologies like OVM and VMM make heavy use of the factory concept. The factory method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The factory method design pattern handles this problem by defining a separate method for creating the objects, whose subclasses can then override to specify the derived type of product that will be created. More generally, the term factory method is often used to refer to any method whose main purpose is creation of objects.

Or in simple terms factory pattern help in creation of the object when you dont know the exact type of the object. the normal way of creating the object is :

```
// Normal Type based object creation

// Class object
class my_class;
int i;
endclass

program main;
// Create object type my_class
my_class obj1;
obj1 = new
endprogram

// Using Factory I should be able to do the following

program main;
base_class my_class_object;

base_class = factory.create_object("my_class"); // See here the type of the object to be created is passed as a string so we dont know the exact type of the object
endprogram
```

• Explain The Difference Between Data Types Logic And Reg And Wire ?

Wire are Reg are present in the verilog and system verilog adds one more data type called logic.

Wire : Wire data type is used in the continuous assignments or ports list. It is treated as a wire So it can not hold a value. It can be driven and read. Wires are used for connecting different modules.

Reg : Reg is a date storage element in system verilog. Its not a actual hardware register but it can store values. Register retain there value until next assignment statement.

Logic : System verilog added this additional datatype extends the rand eg type so it can be driven by a single driver such as gate or module. The main difference between logic dataype and reg/wire is that a logic can be driven by both continuous assignment or blocking/non blocking assignment.

• What Is The Need Of Clocking Blocks ?

- It is used to specify synchronization characteristics of the design
- It Offers a clean way to drive and sample signals
- Provides race-free operation if input skew > 0
- Helps in testbench driving the signals at the right time
- Features

    - Clock specification
    - Input skew,output skew
    - Cycle delay (##)

- Can be declared inside interface,module or program

Example :

```
Module M1(ck, enin, din, enout, dout);
input         ck,enin;
input  [31:0] din    ;
output        enout  ;
output [31:0] dout   ;

clocking sd @(posedge ck);
input  #2ns ein,din    ;
output #3ns enout, dout;
endclocking:sd

reg [7:0] sab ;
initial begin
sab = sd.din[7:0];
end
```

16.endmodule:M1

- **What Are The Ways To Avoid Race Condition Between Testbench And Rtl Using Systemverilog?**

There are mainly following ways to avoid the race condition between testbench and RTL using system verilog

- Program Block
- Clocking Block
- Using non blocking assignments.

- **What Are The Types Of Coverages Available In Sv ?**

Using covergroup : variables, expression, and their cross

Using cover keyword : properties

- **What Is Oops?**

Here are some nice OOP links on SystemVerilog OOP which can be used as a good starting point/reference.

- Object Oriented Programming for Hardware Verification
- Improve Your SystemVerilog OOP Skills by Learning Principles and Patterns
- SystemVerilog OOP OVM Feature Summary
- Enhancing SystemVerilog with AOP Concepts (On how to mimic AOP features in OOP, good for guyz coming from e background)
- Testbench.in OOP Tutorial

- **What Is The Need Of Virtual Interfaces ?**

An interface encapsulate a group of inter-related wires, along with their directions (via modports) and synchronization details (via clocking block). The major usage of interface is to simplify the connection between modules.

But Interface can't be instantiated inside program block, class (or similar non-module entity in SystemVerilog). But they needed to be driven from verification environment like class. To solve this issue virtual interface concept was introduced in SV.

Virtual interface is a data type (that implies it can be instantiated in a class) which hold reference to an interface (that implies the class can drive the interface using the virtual interface). It provides a mechanism for separating abstract models and test programs from the actual signals that make up the design. Another big advantage of virtual interface is that class can dynamically connect to different physical interfaces in run time.

- **What Is The Difference Between Mailbox And Queue?**

A queue is a variable-size, ordered collection of homogeneous elements. A Queue is analogous to one dimensional unpacked array that grows and shrinks automatically. Queues can be used to model a last in, first out buffer or first in, first out buffer.

```
// Other data type as reference
// int q[]; dynamic array
// int q[5]; fixed array
// int q[string]; associate array
// include <
// List#(integer) List1;    //

int q[$] = { 2, 4, 8 };
int p[$];
int e, pos;
e = q[0]; // read the first (leftmost) item
e = q[$]; // read the last (rightmost) item
q[0] = e; // write the first item
p = q; // read and write entire queue (copy)
```

A mailbox is a communication mechanism that allows messages to be exchanged between processes. Data can be sent to a mailbox by one process and retrieved by another.

• What Data Structure You Used To Build Scoreboard?

In SV, we use mailbox to get data from different modules and compare the result.

```
class Scoreboard;
mailbox drvr2sb;
mailbox rcvr2sb;

function new(mailbox drvr2sb,mailbox rcvr2sb);
  this.drvr2sb = drvr2sb;
  this.rcvr2sb = rcvr2sb;
endfunction:new

task start();
  packet pkt_rcv,pkt_exp;
  forever
  begin
    rcvr2sb.get(pkt_rcv);
    $display(" %0d : Scorebooard : Scoreboard received a packet from receiver ",$time);
    drvr2sb.get(pkt_exp);
    if(pkt_rcv.compare(pkt_exp))
    $display(" %0d : Scoreboardd :Packet Matched ",$time);
    else
      $root.error++;
  end
endtask : start
endclass
```

In VMM, we use channels to connect all the modules and compare the result.

```
class Scoreboard extends vmm_xactor;
   Packet_channel   drvr2sb_chan;
   Packet_channel   rcvr2sb_chan;

function new(string inst = "class",
             int unsigned stream_id = -1,
             Packet_channel   drvr2sb_chan = null,
             Packet_channel   rcvr2sb_chan = null);
      super.new("sb",inst,stream_id);
          if(drvr2sb_chan == null)
           `vmm_fatal(this.log,"drvr2sb_channel is not constructed");
      else
           this.drvr2sb_chan = drvr2sb_chan;
            if(rcvr2sb_chan == null)
           `vmm_fatal(this.log,"rcvr2sb_channel is not constructed");
      else
           this.rcvr2sb_chan = rcvr2sb_chan;
          `vmm_note(log,"Scoreboard created ");
endfunction:new

task main();
  Packet pkt_rcv,pkt_exp;
  string msg;
  super.main();
  forever
  begin
    rcvr2sb_chan.get(pkt_rcv);
    $display(" %0d : Scoreboard : Scoreboard received a packet from receiver ",$time);
    drvr2sb_chan.get(pkt_exp);
    if(pkt_rcv.compare(pkt_exp,msg))
    $display(" %0d : Scoreboard :Packet Matched ",$time);
    else
    `vmm_error(this.log,$psprintf(" Packet MissMatched n %s ",msg));
  end
endtask : main
endclass
```

- **What Is The Difference Between $random() And $urandom()?**

- $random system function returns a 32-bit signed random number each time it is called
- $urandom system function returns a 32-bit unsigned random number each time it is called. (newly added in SV, not present in verilog)

What Is Scope Randomization?

Scope randomization ins SystemVerilog allows assignment of unconstrained or constrained random value to the variable within current scope

```
module MyModule;
integer var, MIN;

initial begin
MIN = 50;
for ( int i = 0;i begin
if( randomize(var) with { var < 100 ; var > MIN ;})
 $display(" Randomization sucsessfull : var = %0d Min = %0d",var,MIN);
else
$display("Randomization failed");
end

$finish;
end
endmodule
```

List The Predefined Randomization Methods.

- randomize
- pre_randomize
- post_randomize

What Is The Dfference Between Always_combo And Always@(*)?

From SystemVerilog LRM 3.1a:-

- always_comb get executed once at time 0, always @* waits till a change occurs on a signal in the inferred sensitivity list
- Statement within always_comb can't have blocking timing, event control, or fork-join statement. No such restriction of always @*
- Optionally EDA tool might perform additional checks to warn if the behavior within always_comb procedure doesn't represent combinatorial logic
- Variables on the left-hand side of assignments within an always_comb procedure, including variables from the contents of a called function, shall not be written to by any other processes, whereas always @* permits multiple processes to write to the same variable.
- always_comb is sensitive to changes within content of a function, whereas always @* is only sensitive to changes to the arguments to the function.

A small SystemVerilog code snippet to illustrate #5

```
module dummy;
```

```
logic a, b, c, x, y;

// Example void function
function void my_xor;
input a;        // b and c are hidden input here
x = a ^ b ^ c;
endfunction : my_xor

function void my_or;
input a;        // b and c are hidden input here
y = a | b | c;
endfunction : my_xor

always_comb         // equivalent to always(a,b,c)
my_xor(a);      // Hidden inputs are also added to sensitivity list

always @*         // equivalent to always(a)
my_or(a);     // b and c are not added to sensitivity list
endmodule
```

- **What Is The Use Of Packages?**

In Verilog declaration of data/task/function within modules are specific to the module only. They can't be shared between two modules. Agreed, we can achieve the same via cross module referencing or by including the files, both of which are known to be not a great solution.

The package construct of SystemVerilog aims in solving the above issue. It allows having global data/task/function declaration which can be used across modules. It can contain module/class/function/task/constraints/covergroup and many more declarations (for complete list please refer section 18.2 of SV LRM 3.1a)

The content inside the package can be accessed using either scope resolution operator (::), or using import (with option of referencing particular or all content of the package).

```
package ABC;
// Some typedef
typedef enum {RED, GREEN, YELLOW} Color;

// Some function
void function do_nothing()
...
endfunction : do_nothing

// You can have many different declarations here
endpackage : ABC
```

```

// How to use them
import ABC::Color;   // Just import Color
import ABC::*;     // Import everything inside the package
```

- **What Is The Use Of $cast?**

Type casting in SV can be done either via static casting (', ', ') or dynamic casting via $cast task/function. $cast is very similar to dynamic_cast of C++. It checks whether the casting is possible or not in run-time and errors-out if casting is not possible.

- **How To Call The Task Which Is Defined In Parent Object Into Derived Class ?**

super.task_name();

- **What Is The Difference Between Rand And Randc?**

rand - Random Variable, same value might come before all the the possible value have been returned. Analogous to throwing a dice.

randc - Random Cyclic Variable, same value doesn't get returned until all possible value have been returned. Analogous to picking of card from a deck of card without replacing. Resource intensive, use sparingly/judiciously

- **What Is $root?**

$root refers to the top level instance in SystemVerilog

```
package ABC;
$root.A;     // top level instance A
$root.A.B.C; // item C within instance B within top level instance A
```

- **What Are Bi-directional Constraints?**

Constraints by-default in SystemVerilog are bi-directional. That implies that the constraint solver doesn't follow the sequence in which the constraints are specified. All the variables are looked simultaneously. Even the procedural looking constrains like if ... else ... and -> constrains, both if and else part are tried to solve concurrently. For example (a==0) -> (b==1) shall be solved as all the possible solution of (!(a==0) || (b==1)).

- **What Is Solve And Before Constraint ?**

In the case where the user want to specify the order in which the constraints solver shall solve the constraints, the user can specify the order via solve before construct. i.e.

```
...
constraint XYZ  {
a inside {[0:100]|;
b < 20;
5.a + b > 30;
6.solve a before b;
7.}
```

The solution of the constraint doesn't change with solve before construct. But the probability of choosing a particular solution change by it.

- **Without Using Randomize Method Or Rand,generate An Array Of Unique Values?**

```
1....
2.int UniqVal[10];
3.foreach(UniqVal[i]) UniqVal[i] = i;
4.UniqVal.shuffle();
5....
```

- **Explain About Pass By Ref And Pass By Value?**

Pass by value is the default method through which arguments are passed into functions and tasks. Each subroutine retains a local copy of the argument. If the arguments are changed within the subroutine declaration, the changes do not affect the caller.

In pass by reference functions and tasks directly access the specified variables passed as arguments.Its like passing pointer of the variable.

example:

```
task pass(int i)    //  task pass(var int i) pass by reference
{
delay(10);
i = 1;
printf(" i is changed to %d at %dn",i,get_time(LO) );
delay(10);
i = 2;
printf(" i is changed to %d at %dn",i,get_time(LO) );
}
```

- **What Is The Difference Between Byte And Bit [7:0]?**

byte is signed whereas bit [7:0] is unsigned.

- **What Is The Difference Between Program Block And Module ?**

Program block is newly added in SystemVerilog. It serves these purposes

- It separates testbench from DUT
- It helps in ensuring that testbench doesn't have any race condition with DUT
- It provides an entry point for execution of testbench

- It provides syntactic context (via program ... endprogram) that specifies scheduling in the Reactive Region.

Having said this the major difference between module and program blocks are

- Program blocks can't have always block inside them, modules can have.
- Program blocks can't contain UDP, modules, or other instance of program block inside them. Modules don't have any such restrictions.
- Inside a program block, program variable can only be assigned using blocking assignment and non-program variables can only be assigned using non-blocking assignments. No such restrictions on module
- Program blocks get executed in the re-active region of scheduling queue, module blocks get executed in the active region
- A program can call a task or function in modules or other programs. But a module can not call a task or function in a program.

• What Is The Use Of Modports ?

Modports are part of Interface. Modports are used for specifing the direction of the signals with respect to various modules the interface connects to.

- ...
- interface my_intf;
- wire x, y, z;
- modport master (input x, y, output z);
- modport slave (output x, y, input z);

• Write A Clock Generator Without Using Always Block.

Use of forever begin end. If it is a complex always block statement like always (@ posedge clk or negedge reset_)

```
always @(posedge clk or negedge reset_) begin

  if(!reset_) begin
     data <= '0;
  end else begin
     data <= data_next;
  end
end
```

```
// Using forever : slightly complex but doable

forever begin
  fork
  begin : reset_logic
     @ (negedge reset_);
     data <= '0;
  end : reset_logic
  begin : clk_logic
     @ (posedge clk);
     if(!reset_)    data <= '0;
     else           data <= data_next;
  end : clk_logic
  join_any
  disable fork
end
```

- **What Is Circular Dependency And How To Avoid This Problem ?**

Over specifying the solving order might result in circular dependency, for which there is no solution, and the constraint solver might give error/warning or no constraining. Example

```
1....
2.int x, y, z;
3.constraint XYZ  {
4.solve x before y;
5.solve y before z;
6.solve z before x;
7.....
8.}
```

- **What Is Cross Coverage ?**

Queue has a certain order. It's hard to insert the data within the queue. But Linkedlist can easily insert the data in any location.

- **How To Randomize Dynamic Arrays Of Objects?**

```
class ABC;
// Dynamic array
rand bit [7:0] data [];
// Constraints
constraint cc {
// Constraining size
data.size inside {[1:10]};
// Constraining individual entry
data[0] > 5;
// All elements
foreach(data[i])
if(i > 0)
data[i] > data[i-1];|
}
endclass : ABC
```

What Is The Need Of Alias In Sv?

The Verilog has one-way assign statement is a unidirectional assignment and can contain delay and strength change. To have bidirectional short-circuit connection SystemVerilog has added alias statement.

What Is "this"?

"this" pointer refers to current instance.

What Is Tagged Union ?

An union is used to stored multiple different kind/size of data in the same storage location.

```
typedef union{
bit [31:0]  a;
int         b;
} data_u;
```

Now here XYZ union can contain either bit [31:0] data or an int data. It can be written with a bit [31:0] data and read-back with a int data. There is no type-checking done.

In the case where we want to enforce that the read-back data-type is same as the written data-type we can use tagged union which is declared using the qualifier tagged. Whenever an union is defined as tagged, it stores the tag information along with the value (in expense of few extra bits). The tag and values can only be updated together using a statically type-checked tagged union expression. The data member value can be read with a type that is consistent with current tag value, making it impossible to write one type and read another type of value in tagged union. (the details of which can be found in section 3.10 and 7.15 of SV LRM 3.1a).

```
typedef union tagged{
bit [31:0]  a;
int         b;
} data_tagged_u;

// Tagged union expression
data_tagged_u data1 = tagged a 32'h0;
data_tagged_u data2 = tagged b 5;

// Reading back the value
int xyz = data2.b;
```

What Is "scope Resolution Operator"?

extern keyword allows out-of-body method declaration in classes. Scope resolution operator (::) links method declaration to class declaration.

```
class XYZ;
```

```
// SayHello() will be declared outside the body
// of the class
extern void task SayHello();
endclass : XYZ
void task XYZ :: SayHello();
$Message("Hello !!!n");
endtask : SayHello
```

- **What Is The Difference Between Bits And Logic?**

bits is 2-valued (1/0) and logic is 4-valued (0/1/x/z)

- **What Is The Difference Between $rose And Posedge?**

posedge return an event, whereas $rose returns a Boolean value. Therefore they are not interchangeable.

- **What Is Layered Architecture ?**

In SystemVerilog based constrained random verification environment, the test environment is divided into multiple layered as shown in the figure. It allows verification component re-use across verification projects.

- **What Is The Difference Between Initial Block And Final Block?**

There are many difference between initial and final block. I am listing the few differences that is coming to mind now.

- The most obvious one : Initial blocks get executed at the beginning of the simulation, final block at the end of simulation
- Final block has to be executed in zero time, which implies it can't have any delay, wait, or non-blocking assignments. Initial block doesn't have any such restrictions of execution in zero time (and can have delay, wait and non-blocking statements)

Final block can be used to display statistical/genaral information regarding the status of the execution like this:-

```
1.final begin
2.$display("Simulation Passed");
3.$display("Final value of xyz = %h",xyz);
4.$display("Bye :: So long, and Thanks for all the fishes");
5.end
```

- **How To Check Weather A Handles Is Holding Object Or Not ?**

It is basically checking if the object is initialized or not. In SystemVerilog all uninitialized object handles have a special value of null, and therefore whether it is holding an object or not can be found out by comparing the object handle to null. So the code will look like:-

```
usb_packet My_usb_packet;
...
if(My_usb_packet == null) begin
// This loop will get exited if the handle is not holding any object
....
end else begin
// Hurray ... the handle is holding an object
...
end
```