**Overview on functional verification using system Verilog and Reasons on why we use SV over Verilog?**

1. Why system Verilog over Verilog?

* System Verilog has more language built-in constructs compared to Verilog, which makes it easier for the testbench development using sv.
* There are many advancements in arrays, data types etc. in sv compared to Verilog.
* In Verilog, if we want to reuse the testbench for other designs, the only things to use are functions and tasks. But in sv, we can use classes with little modifications.

1. The primary objective of functional verification engineer is to check the design functionality against the design specifications- catching as many bugs as possible.
2. Verification happens at only 2 steps in the whole VLSI Design flow.

* Functional verification after development of RTL.
* Post silicon validation after we are done with the fabrication- Detecting bugs at this stage is very expensive- We have to repeat the design cycle again.

1. Driving factors of functional verification.

* Less time in developing the testbench environment.
* More time in developing testcases and debugging failing testcases- This is the reason why we use sv for verification.
* We can design testbench for any complex design using Verilog, but it will take lot of time because of having less number of constructs. Also, it has no modularity (ability to develop small units for dedicated functions like classes-gen, driver etc.) and reusability.
* If we create testbench using Verilog also, we can connect it to DUT using port-to-port connection which is very difficult. Also, Most of the TB goes into a single module- Task, functions were used to make code resuable.
* Tasks and functions are the only and only things that we can use in Verilog to create Testbenches.
* We can create separate modules for their dedicated functions using object oriented programming in system Verilog (gen, drv, mon, checker, reference model, coverage, scoreboard).
* In SV Testbenches, Because of modularity, one component implementation Issues doesn’t spoil other components.
* Object oriented programming is the master here which makes it happen for modularity and reusability.

1. When we have sv, why uvm?

* In system Verilog, we have to develop every component of testbench from scratch using OOPs.
* But in UVM, it has pre-defined base classes. We can just extend those pre-defined classes using inheritance and polymorphism as per our needs.
* So, it takes even less time for Testbench development using UVM compared to SV. So, Now-a-days companies are moving towards UVM for Testbench development.
* UVM is not a language. It is a framework built on top of using system Verilog constructs.
* So, we can’t skip SV and learn UVM. The language and constructs that we in UVM are all related to system Verilog only

1. If someone gave me a Testbench architecture diagram and ask me to develop in Verilog- It is very difficult because, it doesn’t support TB architecture with a lot of TB components. The major reason is connecting thos components at port level is very difficult to do.

* So, we use SV to develop modular testbenches – Kind of TB developing style where whole functionality is divided into multiple components.

1. **By product of modularity is REUSABILITY –** We can develop testbenches for new designs using existing TB components.
2. **In summary,** For simple designs, developing TB using Verilog is easier. And for medium and complex designs, using SV and UVM is better for testbench development.
3. **When we close the Functional verification?**

* We we have 100% Testcases passed, 100% functional and code coverage, and 100% Assertion coverage.

**------------------------------------------------------------------------------------------------------------------**

**SV BASIC LANGUAGE CONCEPTS – How do they differ from Verilog?**

**DATA TYPES: 2 types**

* **Static Data types:** Verilog only supports this. So it is not efficient for memory utilization.
* **Dynamic Data type:** system Verilog supports both static and Dynamic data types. We can allocate memory at run time using new constructor for dynamic data types. So, it is very efficient for memory utilization.

**Example** :

Ethernet frame will have a length in between 42 and 1500 octets.

* If I want to declare it in **Verilog**, I need to declare the vector for the maximum possible size 🡪 reg [7:0] payload [1499:0];
* Whereas in **SV**, we can declare a dynamic array, and then later we can allaocate space for only for what is required
* 🡪 reg [7:0] payload [$]; //QUEUE
* 🡪reg [7:0] payload []; //DYNAMIC ARRAY
* 🡪 reg [7:0] payload [\*]; //ASSOCIATIVE ARRAY
* Later When I want to create a frame of 100 octet payload size.
* Payload = new[100]; //only 100 element space will be used.
* Payload.delete(); //Whole space gets freed up
* This is main drawback in Verilog, where space of whole array will be used-Since, Verilog has only static data types, it is not efficient for memory utilization.
* SV has mix of both data types, which is very efficient for memory utilization.
* Whenever we work with dynamic arrays, we need to explicitly tell how much memory to allocate.

**Different Phases of Data type usage:**

1. **DEFNITION and INSTANTIATION:**

Language provided: integer a; //it is a default size of 32-bit – language provided

User-defined : type reg [3:0] nibble; //Here, nibble is a user-defined data type created

//Default size of ‘nibble’ data type is 4 bits.

nibble a; // variable ‘a’ is of data type nibble and size is 4 bit vector.

1. **CREATION:**

* For static data types, creation of memory is not required. Since, the memory equal to default size of data type is already created.

Example:

integer a[10];🡪 integer a[9:0] //already 32 bits of memory allocated for 10 elements

a = new(); //not required – static data type

byte b[10]; //already 8 bit vector of memory allocated for 10 elements= 80 bits

b = new(); //No need to allocate the memory again

* For Dynamic Data types, creation of memory is required. Since, we don’t declare elements in advance.
* integer intA[]; //integer is 32-bit vector. Since, it is dynamic array, the array size is zero. We need to create the memory, as per our needs.
* intA = new[5]; //it will create array of size 5 elements. Each element is 32 bit vector.

TWO TYPES OF ‘new’ CONSTRUCTORS:

* ‘new[SIZE]’ constructor is used for allocating memory equal to ‘SIZE’ defined.
* ‘new()’ constructor is used for allocating memory to create object for a class.

1. **RANDOMIZATION:**

* In Verilog, for the purpose of randomizing the variables, we only use **$random, $urandom\_range(10,40).** This is the maximum thing, that we can do in verilog
* No pre-implemented methods/constructs in verilog
* But in SV, we can radaomize variables using
* **$random;**
* **$urandom\_range(10,40);**
* **Randcase**
* **Randomize //pre-implemented method in SV**
* This is one advantage why we moving from Verilog to SV.

**------------------------------------------------------------------------------------------------------------------**

1. As discussed above, Verilog has no pre-implemented methods. For every thing, we need to develop it from scratch.
2. But, In Sv, there are some many built-in methods(Pre-implemented constructs), which makes life more easier for constructing complex testbenches.

Example:

* If we want to use FIFO in Verilog, we need to implement the module and we can use that by instantiating that module.
* But, in SV, there is a built-in method called MAILBOX(~similar to FIFO). rWe can directly instantiate mailbox with a name and then we can use it.
* In Verilog, for FIFO depth, we can use parameter over-riding concept. Similarly, here in Verilog we can define depth of mailbox like:
* mailbox mbox = new(100); //mbox with depth 100 equivalent memory gets created.

1. We can not use mailbox in designs. Because it is non-synthesizable.
2. So, SV provides lots of built-in constructs and also provides built-in methods to use those constructs.
3. We only need to how how to use those constructs, and we don’t care about how it is internally created.