Introduction:

- What is system Verilog?
- Why we go for System Verilog?
- What is verification?
- How to verify Design?
- Why system Verilog for verification?
- Evolution of System Verilog
- System Verilog Features
- Data Types

What is System Verilog?

- System Verilog is a hardware description and hardware verification language.
- System Verilog is used to model, design, simulate, test and implement electronic systems.
- System Verilog is based on Verilog and some extensions.

What is Hardware Description Language?

Hardware description language (HDL) is a specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits.

What is Hardware Verification Language?

Hardware verification language(HVL) is a programming language used to verify the designs of electronic circuits written in a HDL.

Why we go for System Verilog?

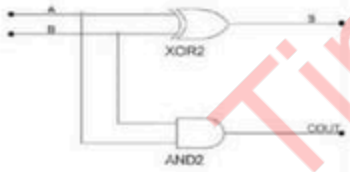
- Verilog was the primary language to verify functionality of designs that were small, not very complex and had less features.
- As design complexity increase, so does the requirement of better tools to design and verify it.
- System Verilog is far superior to Verilog because of its ability to perform constraint random stimulus, use oops features in testbench construction, functional coverage, assertions among many others.

What is verification?

- Verification is the process of ensuring that a given hardware design works as expected.
- Make sure design is bug free by verifying it in all aspects. All possible scenarios.

Example:

Schematic Half Adder



Verilog Half Adder

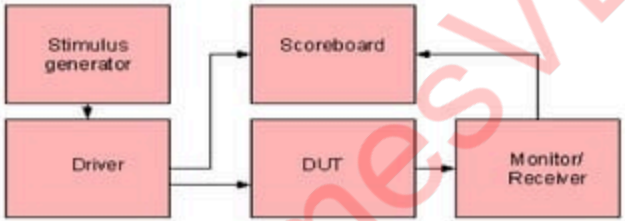
```
module add_half (a, b, s, cout);

input  a, b;
output s, cout;
wire s, cout;

assign s = a ^ b;
assign cout = a & b;

endmodule   // end of half adder module
```

How to verify DUT?



Why system Verilog for verification?

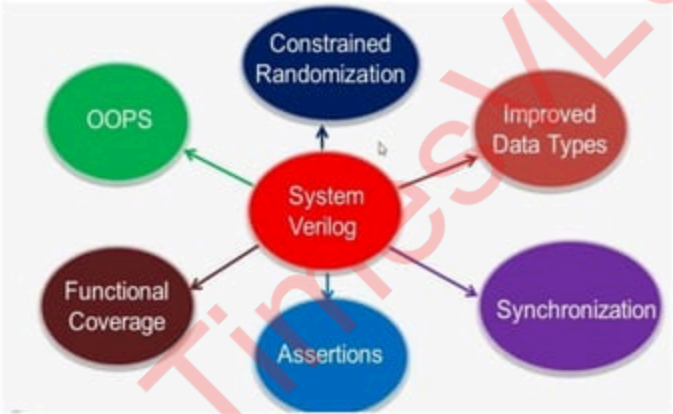
- Verilog was initially used for writing testbench.
- But, writing complex testbench is much more of programming task than describing hardware.
- No need to synthesize testbench.
- Now UVM is replacing SV based verification in industry.
- Still, UVM = Structured SV. knowing SV based verification helps understanding UVM based verification, else UVM feels like set of magic macros.

Evaluation of SV:

System Verilog language components are:

- Concepts of Verilog HDL.
- Testbench constructs based on Vera.
- Open Vera assertions.
- Synopsys' VCS DirectC simulation interface to C and C++.
- A coverage application programming interface that provides links to coverage metrics.

System Verilog Features:



Data Types:

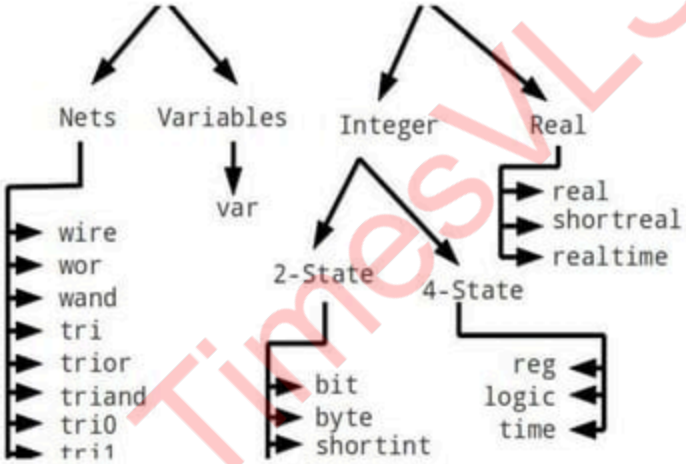
What is data type?

- In programming, data type is a classification that specifies which type of value a variable has.
- For example, A string, is a data type that is used to classify text.
- An integer is a data type used to classify whole numbers.

Data Types:

- SystemVerilog added lot of new data types and improved the existing data types to improve run time memory utilization of simulators.
- In System Verilog data types can be classified into 2-state types and 4-state types.
- 2-state types can take only 0, 1, where as 4-state types can take 0,1,X,Z.
- 2-state types consume less (50%) memory and simulate faster when compared to 4-state types.
- SV introduces a new 4-state data type called logic that can be driven in both procedural blocks and continuous assign statements.
- But, a signal with more than one driver needs to be declared a net-type such as wire so that System Verilog can resolve the final value.

Data Types:



Data Types:

Data Types				
Type	**Mode**	**State**	**Size**	**Sign**
bit	integer	2-state	1-bit	unsigned
bit [x:0]	integer	2-state	user defined	unsigned
byte	integer	2-state	8-bit	signed
shortint	integer	2-state	16-bit	signed
int	integer	2-state	32-bit	signed
longint	integer	2-state	64-bit	signed
logic	integer	4 state	user defined	unsigned
reg	integer	4 state	user defined	unsigned
integer	integer	4 state	32-bit	signed
time	integer	4 state	64-bit	unsigned
real	floatingpoint	2-state	-	-

Void Data Type:

- void is used in functions to return no value.
- Void data type represents non-existent data.
- This type can be specified as the return type of function, including no return value.

Syntax:

```
function void display ();
$display ("Am not going to return any value");
endfunction
```

Arrays:

- An array is a collection of variables, all of the same type, and accessed using the same name plus one or more indices.
- There are different types of arrays:

Examples:

```
int array1 [6];          //fixed size single dimension array
int array2 [5:0];        //fixed size single dimension array
int array3 [3:0][2:0];   //fixed size multi dimension array
bit [7:0] array4[2:0];   //unpacked array declaration
bit [2:0][7:0] array5;   //packed array declaration
bit [2:0][7:0] array6 [3]; //mixed packed and unpacked array
```

Fixed array:

- In fixed size array, array size will be constant throughout the simulation,
- Once the array is declared no need to create it.
- By default, the array will be initialized with value '0'.

Single Dimension array:

int array1 [6]; //Compact declaration

int array2 [5:0]; // Verbose declaration

Two Dimension array:

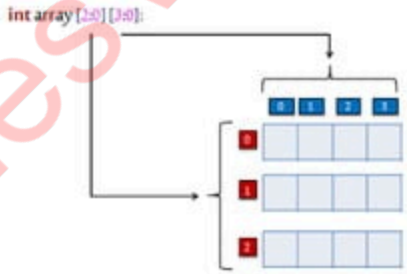
int arr[2][3];

Three Dimension array:

int arr[2][2][2];

Array assignment:

array = '{ '{0,1,2,3},'{4,5,6,7},'{8,9,10,11}};



Packed Array:

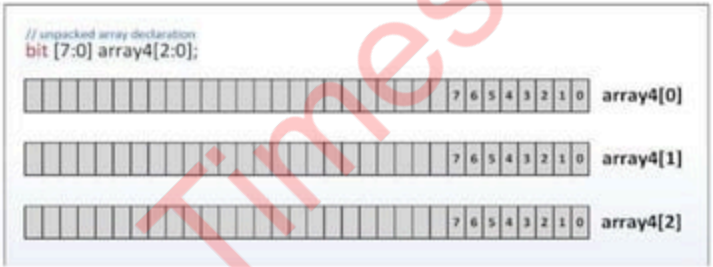
- The term packed array is used to refer to the dimensions declared before the data identifier name.
- Packed arrays can be of single bit data types (reg, logic, bit), enumerated types.
- A packed array is guaranteed to be represented as a contiguous set of bits.

Example:

Unpacked array:

- The term unpacked array is used to refer to the dimensions declared after the data identifier name.
- Unpacked arrays can be of any data type.
- An unpacked array **may** or **may not** be so represented as a **contiguous** set of bits.

Example:



Dynamic array:

- A dynamic array is one dimension of an unpacked array whose size can be set or changed at run-time.
- Dynamic array is Declared using an empty word subscript [ ].
- The space for a dynamic array doesn't exist until the array is explicitly created at run-time, space is allocated when new[number] is called.
- The number indicates the number of space/elements to be allocated.
- Dynamic arrays are useful for contiguous collections of variables whose number changes dynamically.

Syntax:

Data_type array_name[];

Methods:

new[ ] -> allocates the storage.
size( ) -> returns the current size of a dynamic array.
delete( ) -> empties the array, resulting in a zero-sized array.

Dynamic array:

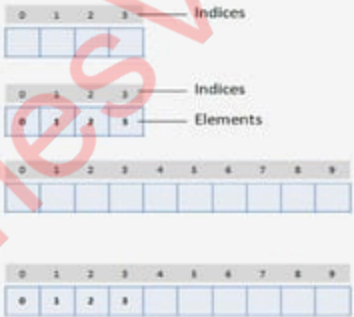
```
//dynamic array declaration
bit [7:0] d_array[];

//memory allocation
d_array = new[4];

//array initialization
d_array = {0,1,2,3};

// Increasing the size by overriding the old values
 d_array = new[10];

//Increasing the size by retaining the old values
d_array = new[10](d_array);
```



Associative Array:

- Associative array Stores entries in a sparse matrix.
- Associative arrays allocate the storage only when it is used, unless like in the dynamic array we need to allocate memory before using it.
- In associative array index expression is not restricted to integral expressions, but can be of any type.
- When the size of the collection is unknown or the data space is sparse, an associative array is a better option.

Syntax:

```
data_type array_name [ index_type ];
```

Associative array:

Examples:

```
int a_array1[*] ;            // associative array of integer (unspecified index)
bit [31:0] a_array2[string]; // associative array of 32-bit, indexed by string
ev_array [myClass];          //associative array of event, indexed by class
```

Associative array:

Methods:

Method	Description
num()	returns the number of entries in the associative array
delete(index)	removes the entry at the specified index.exa_array.delete(index)
exists(index)	returns 1 if an element exists at the specified index else returns 0
first(var)	assigns the value of first index to the variable var
last(var)	assigns the value of last index to the variable var
next(var)	assigns the value of next index to the variable var
prev(var)	assigns the value of previous index to the variable var

Queue:

- A queue is a variable-size, ordered collection of homogeneous elements.
- Like a dynamic array, queues can grow and shrink.
- Queue supports adding and removing elements anywhere.
- Queues are declared using the same syntax as unpacked arrays, but specifying $ as the array size. In queue 0 represents the first, and $ representing the last entries.

Syntax:

Data_type queue_name[$];

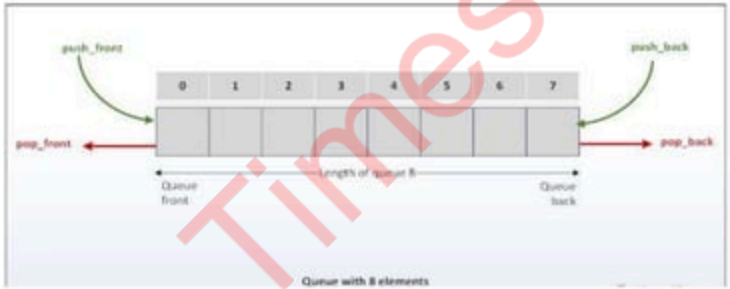
Example:

```
bit    queue_1[$];    // queue of bits  (unbound queue)
int    queue_2[$];    // queue of int
byte   queue_3[$:255]; // queue of byte (bounded queue with 256 entries)
string queue_4[$];    // queue of strings
```

Queue:

- A queue can be bounded or unbounded.

  bounded queue - queue with the number of entries limited or queue size specified.



Queue:

unbounded queue - queue with unlimited entries or queue size not specified.

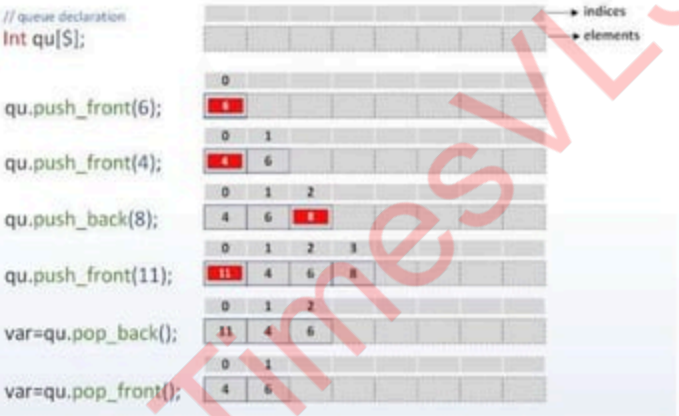
Queue:

Queue Methods

Method	Description
size()	returns the number of items in the queue
insert()	inserts the given item at the specified index position
delete()	deletes the item at the specified index position
push_front()	inserts the given element at the front of the queue
push_back()	inserts the given element at the end of the queue
pop_front()	removes and returns the first element of the queue
pop_back()	removes and returns the last element of the queue

Queue:

```
// queue declaration
Int qu[$];
```



Events:

- Events are static objects useful for synchronization between the process.
- Events operations are of two staged processes in which one process will trigger the event, and the other processes will wait for an event to be triggered.
- Events are triggered using -> operator or ->> operator
- wait for an event to be triggered using @ operator or wait() construct
- System Verilog events act as handles to synchronization queues. thus, they can be passed as arguments to tasks, and they can be assigned to one another or compared.

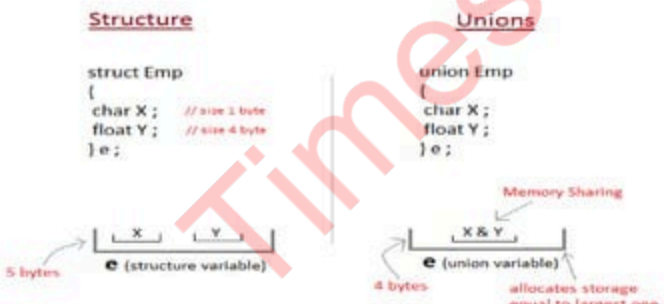
Syntax:

```
->event_name;
@(event_name.triggered);
```

Structure & Union:

- A structure is a user-defined data type. that allows to combining data items of different kinds. Structures are used to represent a record.
- Like Structures, union is a user defined data type. In union, all members share the same memory location.

Example:



Classes:

- A class is a user-defined data type.
- Classes consist of data (called *properties*) and tasks and functions to access the data (called *methods*).
- Classes are used in object-oriented programming.
- In SystemVerilog, classes support the following aspects of object-orientation – encapsulation, data hiding, inheritance and polymorphism.

Example:

```
class c;
    int x;
    task set (int i);
    x=1;
    endtask;
endmodule;
```