

# SYSTEM VERILOG

## INTRODUCTION

SystemVerilog is a combined Hardware Description Language and Hardware Verification Language based on extensions to Verilog. SystemVerilog was created by the donation of the Superlog language to Accellera in 2002. The bulk of the verification functionality is based on the OpenVera language donated by Synopsys. In 2005, SystemVerilog was adopted as IEEE Standard 1800-2005 .Few of SystemVerilog's capabilities are unique, but it is significant that these capabilities are combined and offered within a single HDL. There is great value in a common HDL which handles all aspects of the design and verification flow: design description, functional simulation, property specification, and formal verification.

**DATA TYPES** :SystemVerilog adds extended and new data types to Verilog for better encapsulation and compactness. SystemVerilog extends Verilog by introducing C like data types. SystemVerilog adds a new 2-state data types that can only have bits with 0 or 1 values unlike verilog 4-state data types which can have 0, 1, X and Z. SystemVerilog also allows user to define new data types. SystemVerilog offers several data types, representing a hybrid of both Verilog and C data types. SystemVerilog 2-state data types can simulate faster, take less memory, and are preferred in some design styles. Then a 4-state value is automatically converted to a 2-state value, X and Z will be converted to zeros.

Type	2-state / 4-state	Signed / Unsigned	Number of bits	SystemVerilog / Verilog
<b>shortint</b>	2	Signed	16	SystemVerilog
<b>int</b>	2	Signed	32	SystemVerilog
<b>longint</b>	2	Signed	64	SystemVerilog
<b>byte</b>	2	Signed	8	SystemVerilog
<b>bit</b>	2	Unsigned		SystemVerilog
<b>logic</b>	4	Unsigned		SystemVerilog
<b>reg</b>	4	Unsigned		Verilog
<b>integer</b>	4	Signed	$\geq 32$	Verilog
<b>real</b>				Verilog
<b>shortreal</b>				SystemVerilog
<b>realtime</b>				Verilog
<b>time</b>	4	Unsigned	64	Verilog

**TIP :** If you don't need the x and z values then use the SystemVerilog int and bit types which make execution faster.

### **Signed And Unsigned :**

Integer types use integer arithmetic and can be signed or unsigned. The data types byte, shortint, int, integer, and longint default to signed. The data types bit, reg, and logic default to unsigned, as do arrays of these types.

To use these types as unsigned, user has to explicitly declare it as unsigned.

```
int unsigned ui;  
int signed si  
byte unsigned ubyte;
```

User can cast using signed and unsigned casting.

```
if (signed'(ubyte)< 150) // ubyte is unsigned
```

### **Void :**

The void data type represents nonexistent data. This type can be specified as the return type of functions to indicate no return value.

```
void = function_call();
```

### **LITERALS**

#### **Integer And Logic Literals**

In verilog , to assign a value to all the bits of vector, user has to specify them explicitly.

```
reg[31:0] a = 32'hfffffff;
```

Systemverilog Adds the ability to specify unsized literal single bit values with a preceding ('). '0, '1, 'X, 'Z, 'z // sets all bits to this value.

```
reg[31:0] a = '1;
```

'x is equivalent to Verilog-2001 'bx

'z is equivalent to Verilog-2001 'bz

'1 is equivalent to making an assignment of all 1's

'0 is equivalent to making an assignment of 0

#### **Time Literals**

Time is written in integer or fixed-point format, followed without a space by a time unit (fs ps ns us ms s step).

#### **EXAMPLE**

0.1ns

40ps

The time literal is interpreted as a realtime value scaled to the current time unit and rounded to the current time precision.

#### **Array Literals**

Array literals are syntactically similar to C initializers, but with the replicate operator ( {{}} ) allowed.

## **EXAMPLE**

```
int n[1:2][1:3] = '{'{'0,1,2},'{3{4}}};
```

The nesting of braces must follow the number of dimensions, unlike in C. However, replicate operators can be nested. The inner pair of braces in a replication is removed. A replication expression only operates within one dimension.

## **EXAMPLE:**

```
int n[1:2][1:6] = '{2'{'3{4, 5}}}}; // same as '{'{'4,5,4,5,4,5},{4,5,4,5,4,5}}
```

## **Structure Literals**

Structure literals are structure assignment patterns or pattern expressions with constant member expressions A structure literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context.

## **EXAMPLE**

```
typedef struct {int a; shortreal b;} ab;  
ab c;  
c = '{0, 0.0}; // structure literal type determined from  
// the left-hand context (c)
```

Nested braces should reflect the structure.

## **EXAMPLE**

```
ab abarr[1:0] = '{'{'1, 1.0}, '{2, 2.0}};
```

The C-like alternative '{1, 1.0, 2, 2.0} for the preceding example is not allowed.

## **EXAMPLE: default values**

```
c = '{a:0, b:0.0};  
c = '{default:0};  
d = ab'{int:1, shortreal:1.0};
```

## **STRINGS**

In Verilog, string literals are packed arrays of a width that is a multiple of 8 bits which hold ASCII values. In Verilog, if a string is larger than the destination string variable, the string is truncated to the left, and the leftmost characters will be lost. SystemVerilog adds new keyword "string" which is used to declare string data types unlike verilog. String data types can be of arbitrary length and no truncation occurs.

```
string myName = "TEST BENCH";
```

## **String Methods :**

SystemVerilog also includes a number of special methods to work with strings. These methods use the built-in method notation. These methods are:

1. str.len() returns the length of the string, i.e., the number of characters in the string.
2. str.putc(i, c) replaces the ith character in str with the given integral value.
3. str.getc(i) returns the ASCII code of the ith character in str.
4. str.toupper() returns a string with characters in str converted to uppercase.
5. str.tolower() returns a string with characters in str converted to lowercase.

6. str.compare(s) compares str and s, and return value. This comparison is case sensitive.
7. str.icompare(s) compares str and s, and return value .This comparison is case insensitive.
8. str.substr(i, j) returns a new string that is a substring formed by index i through j of str.
9. str.atoi() returns the integer corresponding to the ASCII decimal representation in str.
10. str.atoreal() returns the real number corresponding to the ASCII decimal representation in str.
11. str.itoa(i) stores the ASCII decimal representation of i into str (inverse of atoi).
12. str.hextoa(i) stores the ASCII hexadecimal representation of i into str (inverse of atohex).
13. str.bintoa(i) stores the ASCII binary representation of i into str (inverse of atobin).
14. str.realtoa(r) stores the ASCII real representation of r into str (inverse of atoreal)

### EXAMPLE : String methods

```
module str;
string A;
string B;
initial
begin
  A = "TEST ";
  B = "Bench";
  $display("%d",A.len());
  $display("%s",A.getc(5));
  $display("%s",A.tolower);
  $display("%s",B.toupper);
  $display("%d",B.compare(A));
  $display("%d",A.compare("test"));
  $display("%s",A.substr(2,3)); A = "111";
  $display("%d",A.atoi());
end
endmodule
```

### RESULTS :

```
5
test
BENCH
-18
-32
ST
111
```

## **String Pattern Match**

Use the following method for pattern matching in SystemVerilog. Match method which is in OpenVera or C , is not available in SystemVerilog . For using match method which is in C , use the DPI calls . For native SystemVerilog string match method, hear is the example.

**CODE:**

```
function match(string s1,s2);
    int l1,l2;
    l1 = s1.len();
    l2 = s2.len();
    match = 0 ;
    if( l2 > l1 )
        return 0;
    for(int i=0;i < l1 - l2 + 1; i++)
        if( s1.substr(i,i+l2 -1) == s2)
            return 1;
endfunction
```

**EXAMPLE:**

```
program main;
    string str1,str2;
    int i;
initial
begin
    str1 = "this is first string";
    str2 = "this";
    if(match(str1,str2))
        $display(" str2 : %s : found in :%s:",str2,str1);
    str1 = "this is first string";
    str2 = "first";
    if(match(str1,str2))
        $display(" str2 : %s : found in :%s:",str2,str1);
    str1 = "this is first string";
    str2 = "string";
    if(match(str1,str2))
        $display(" str2 : %s : found in :%s:",str2,str1);

    str1 = "this is first string";
    str2 = "this is ";
    if(match(str1,str2))
        $display(" str2 : %s : found in :%s:",str2,str1); str1 = "this is first string";
    str2 = "first string";
```

```

if(match(str1,str2))
$display(" str2 : %s : found in :%s:",str2,str1);

str1 = "this is first string";
str2 = "first string "// one space at end
if(match(str1,str2))
    $display(" str2 : %s : found in :%s:",str2,str1);
end
endprogram

```

### **RESULTS:**

str2 : this : found in :this is first string:  
 str2 : first : found in :this is first string:  
 str2 : string : found in :this is first string:  
 str2 : this is : found in :this is first string:  
 str2 : first string : found in :this is first string:

### **String Operators**

SystemVerilog provides a set of operators that can be used to manipulate combinations of string variables and string literals. The basic operators defined on the string data type are

### **Equality**

Syntax : Str1 == Str2

Checks whether the two strings are equal. Result is 1 if they are equal and 0 if they are not. Both strings can be of type string. Or one of them can be a string literal. If both operands are string literals, the operator is the same Verilog equality operator as for integer types.

### **EXAMPLE**

```

program main;
  initial
  begin
    string str1,str2,str3;
    str1 = "TEST BENCH";
    str2 = "TEST BENCH";
    str3 = "test bench";
    if(str1 == str2)
      $display(" Str1 and str2 are equal");
    else
      $display(" Str1 and str2 are not equal");
    if(str1 == str3)
      $display(" Str1 and str3 are equal");
    else
      $display(" Str1 and str3 are not equal");
  
```

```
    end  
endprogram
```

## RESULT

Str1 and str2 are equal  
Str1 and str3 are not equal

### Inequality.

Syntax: Str1 != Str2

Logical negation of Equality operator. Result is 0 if they are equal and 1 if they are not. Both strings can be of type string. Or one of them can be a string literal. If both operands are string literals, the operator is the same Verilog equality operator as for integer types.

## EXAMPLE

```
program main;  
    initial  
    begin  
        string str1,str2,str3;  
        str1 = "TEST BENCH";  
        str2 = "TEST BENCH";  
        str3 = "test bench";  
        if(str1 != str2)  
            $display(" Str1 and str2 are not equal");  
        else  
            $display(" Str1 and str2 are equal");  
        if(str1 != str3)  
            $display(" Str1 and str3 are not equal");  
        else  
            $display(" Str1 and str3 are equal");  
    end  
endprogram
```

## RESULT

Str1 and str2 are equal  
Str1 and str3 are not equal

### Comparison.

Syntax:

Str1 < Str2  
Str1 <= Str2  
Str1 > Str2  
Str1 >= Str2

Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings Str1 and Str2. The comparison uses the compare string method. Both operands can be of type string, or one of them can be a string literal.

### EXAMPLE

```
program main;
initial
begin
    string Str1,Str2,Str3;
    Str1 = "c";
    Str2 = "d";
    Str3 = "e";
    if(Str1 < Str2)
        $display(" Str1 < Str2 ");
    if(Str1 <= Str2)
        $display(" Str1 <= Str2 ");
    if(Str3 > Str2)
        $display(" Str3 > Str2");
    if(Str3 >= Str2)
        $display(" Str3 >= Str2");
end
endprogram
```

### RESULT

```
Str1 < Str2
Str1 <= Str2
Str3 > Str2
Str3 >= Str2
```

### Concatenation.

Syntax: {Str1,Str2,...,Strn}

Each operand can be of type string or a string literal (it shall be implicitly converted to type string). If at least one operand is of type string, then the expression evaluates to the concatenated string and is of type string. If all the operands are string literals, then the expression behaves like a Verilog concatenation of integral types; if the result is then used in an expression involving string types, it is implicitly converted to the string type.

### EXAMPLE

```
program main;
initial
begin
```

```

string Str1,Str2,Str3,Str4,Str5;
Str1 = "WWW.";
Str2 = "TEST";
Str3 = "";
Str4 = "BENCH";
Str5 = ".IN";
$display("%s", {Str1,Str2,Str3,Str4,Str5});
end
endprogram

```

## RESULT

WWW.TESTBENCH.IN

### Replication.

Syntax : {multiplier{Str}}

Str can be of type string or a string literal. Multiplier must be of integral type and can be nonconstant. If multiplier is nonconstant or Str is of type string, the result is a string containing N concatenated copies of Str, where N is specified by the multiplier. If Str is a literal and the multiplier is constant, the expression behaves like numeric replication in Verilog (if the result is used in another expression involving string types, it is implicitly converted to the string type).

## EXAMPLE

```

program main;
  initial
  begin
    string Str1,Str2;
    Str1 = "W";
    Str2 = ".TESTBENCH.IN";
    $display("%s", {{3{Str1}},Str2});
  end
endprogram

```

## RESULT

WWW.TESTBENCH.IN

### Indexing.

Syntax: Str[index]

Returns a byte, the ASCII code at the given index. Indexes range from 0 to N-1, where N is the number of characters in the string. If given an index out of range, returns 0. Semantically equivalent to Str.getc(index)

## EXAMPLE

```

program main;
  initial
  begin
    string Str1;

```

```

Str1 = "WWW.TESTBENCH.IN";
for(int i=0 ;i < 16 ; i++)
    $write("%s ",Str1[i]);
end
endprogram

```

## RESULT

W W W . T E S T B E N C H . I N

## USERDEFINED DATATYPES

Systemverilog allows the user to define datatypes. There are different ways to define user defined datatypes. They are

1. Class.
2. Enumerations.
3. Struct.
4. Union.
5. Typedef.

## ENUMERATIONS

You'll sometimes be faced with the need for variables that have a limited set of possible values that can be usually referred to by name. For example, the state variable like

IDLE,READY,BUZY etc of state machine can only have the all the states defined and Refraining or displaying these states using the state name will be more comfortable. There's a specific facility, called an enumeration in SystemVerilog . Enumerated data types assign a symbolic name to each legal value taken by the data type. Let's create an example using one of the ideas I just mentioned-a state machine .

You can define this as follows:

```
enum {IDLE,READY,BUZY} states;
```

This declares an enumerated data type called states, and variables of this type can only have values from the set that appears between the braces, IDLE,READY and BUZY. If you try to set a variable of type states to a value that isn't one of the values specified, it will cause an error. Enumerated data type are strongly typed.

One more advantage of enumerated datatypes is that if you don't initialize them , each one would have a unique value. By default they are int types. In the previous examples, IDLE is 0, READY is 1 and BUZY is 2. These values can be printed as values or strings.

The values can be set for some of the names and not set for other names. A name without a value is automatically assigned an increment of the value of the previous name. The value of first name by default is 0.

```

// c is automatically assigned the increment-value of 8
enum {a=3, b=7, c} alphabet;
// Syntax error: c and d are both assigned 8
enum {a=0, b=7, c, d=8} alphabet;
// a=0, b=7, c=8
enum {a, b=7, c} alphabet;

```

### Enumerated Methods:

SystemVerilog includes a set of specialized methods to enable iterating over the values of enumerated.

The first() method returns the value of the first member of the enumeration.

The last() method returns the value of the last member of the enumeration.

The next() method returns the Nth next enumeration value (default is the next one) starting from the current value of the given variable.

The prev() method returns the Nth previous enumeration value (default is the previous one) starting from the current value of the given variable.

The name() method returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, the name() method returns the empty string.

### **EXAMPLE : ENUM methods**

```

module enum_method;
  typedef enum {red,blue,green} colour;
  colour c;
  initial
  begin
    c = c.first();
    $display("%s",c.name);
    c = c.next();
    $display("%s",c.name);
    c = c.last();
    $display("%s",c.name);
    c = c.prev();
    $display("%s",c.name);
  end
endmodule

```

### **RESULTS :**

```

red
blue
green
blue

```

## Enum Numerical Expressions

Elements of enumerated type variables can be used in numerical expressions. The value used in the expression is the numerical value associated with the enumerated value.

An enum variable or identifier used as part of an expression is automatically cast to the base type of the enum declaration (either explicitly or using int as the default). A cast shall be required for an expression that is assigned to an enum variable where the type of the expression is not equivalent to the enumeration type of the variable.

**EXAMPLE:**

```
module enum_method;
typedef enum {red,blue,green} colour;
colour c,d;
int i;
initial
begin
$display("%s",c.name());
d = c;
$display("%s",d.name());
d = colour'(c + 1); // use casting
$display("%s",d.name());
i = d; // automatic casting
$display("%0d",i);
c = colour'(i);
$display("%s",c.name());
end
endmodule
```

## **RESULT**

```
red
red
blue
1
blue
```

TIP: If you want to use X or Z as enum values, then define it using 4-state data type explicitly.

```
enum integer {IDLE, XX='x, S1='b01, S2='b10} state, next;
```

## STRUCTURES AND UNIONUNS

### Structure:

The disadvantage of arrays is that all the elements stored in them are to be of the same data type. If we need to use a collection of different data types, it is not possible using an array. When we require using a collection of different data items of different data types we can use a structure. Structure is a method of packing data of different types. A structure is a convenient

method of handling a group of related data items of different data types.

```
struct {  
    int a;  
    byte b;  
    bit [7:0] c;  
} my_data_struct;
```

The keyword "struct" declares a structure to holds the details of four fields namely a,b and c. These are members of the structures. Each member may belong to different or same data type. The structured variables can be accessed using the variable name "my\_data\_struct".

```
my_data_struct.a = 123;  
$display(" a value is %d ",my_data_struct.a);
```

### Assignments To Struct Members:

A structure literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context.

```
my_data_struct = ` {1234,8'b10,8'h20};
```

Structure literals can also use member name and value, or data type and default value.

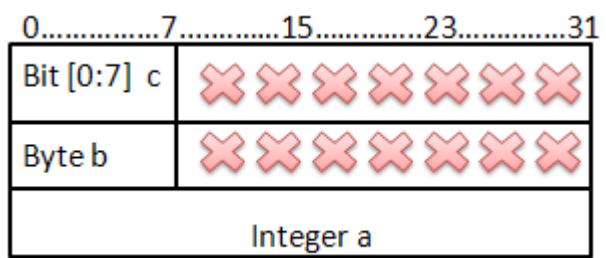
```
my_data_struct = ` {a:1234,default:8'h20};
```

### Union

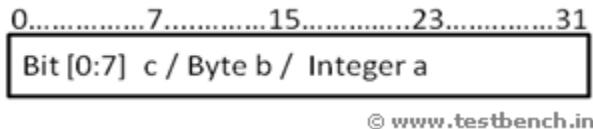
Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area. A union allows us to treat the same space in memory as a number of different variables. That is a Union offers a way for a section of memory to be treated as a variable of one type on one occasion and as a different variable of a different type on another occasion.

```
union {  
    int a;  
    byte b;  
    bit [7:0] c;  
} my_data;
```

memory allocation for the above defined struct "my\_data\_struct".



Memory allocation for the above defined union "my\_data\_union".



© www.testbench.in

### **Packed Structures:**

In verilog , it is not convenient for subdividing a vector into subfield. Accessing subfield requires the index ranges.

For example

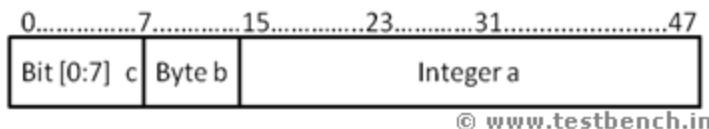
```
reg [0:47] my_data;  
`define a_indx 16:47  
`define b_indx 8:15  
`define c_indx 0:7  
my_data[`b_indx] = 8'b10; // writing to subfield b  
$display("%d",my_data[`a_indx]); // reading subfield a
```

A packed structure is a mechanism for subdividing a vector into subfields that can be conveniently accessed as members. Consequently, a packed structure consists of bit fields, which are packed together in memory without gaps. A packed struct or union type must be declared explicitly using keyword "packed".

```
struct packed {  
    integer a;  
    byte b;  
    bit [0:7] c;  
} my_data;
```

```
my_data.b = 8'b10;  
$display("%d", my_data.a);
```

Memory allocation for the above defined packed struct "my\_data".



© www.testbench.in

One or more bits of a packed structure can be selected as if it were a packed array, assuming an [n-1:0] numbering:

```
My_data [15:8] // b
```

If all members of packed structure are 2-state, the structure as a whole is treated as a 2-state vector.

If all members of packed structure is 4-state, the structure as a whole is treated as a 4-state vector.

If there are also 2-state members, there is an implicit conversion from 4-state to 2-state when reading those members, and from 2-state to 4-state when writing them.

### **TYPEDEF**

A typedef declaration lets you define your own identifiers that can be used in place of type specifiers such as int, byte, real. Let us see an example of creating data type "nibble".

```
typedef bit[3:0] nibble; // Defining nibble data type.  
nibble a, b; // a and b are variables with nibble data types.
```

### **Advantages Of Using Typedef :**

Shorter names are easier to type and reduce typing errors.

Improves readability by shortening complex declarations.

Improves understanding by clarifying the meaning of data.

Changing a data type in one place is easier than changing all of its uses throughout the code.

Allows defining new data types using structs, unions and Enumerations also.

Increases reusability.

Useful is type casting.

Example of typedef using struct, union and enum data types

```
typedef enum {NO, YES} boolean;  
typedef union { int i; shortreal f; } num; // named union type  
typedef struct {  
    bit isfloat;  
    union { int i; shortreal f; } n; // anonymous type  
} tagged_st; // named structure  
  
boolean myvar; // Enum type variable  
num n; // Union type variable  
tagged_st a[9:0]; // array of structures
```

### **ARRAYS**

Arrays hold a fixed number of equally-sized data elements. Individual elements are accessed by index using a consecutive range of integers. Some type of arrays allows to access individual elements using non consecutive values of any data types. Arrays can be classified as fixed-sized arrays (sometimes known as static arrays) whose size cannot change once their declaration is

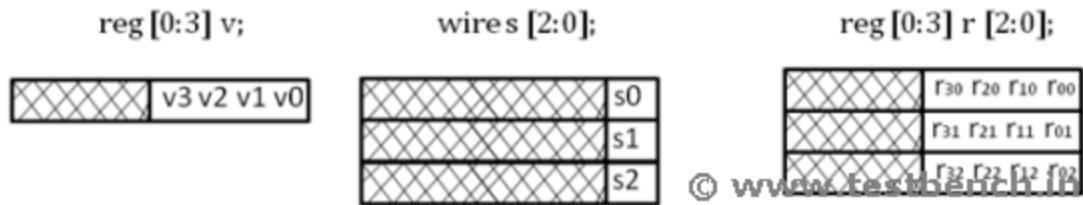
done, or dynamic arrays, which can be resized.

### Fixed Arrays:

"Packed array" to refer to the dimensions declared before the object name and "unpacked array" refers to the dimensions declared after the object name.

SystemVerilog accepts a single number, as an alternative to a range, to specify the size of an unpacked array. That is, [size] becomes the same as [0:size-1].

**int** Array[8][32]; is the same as: **int** Array[0:7][0:31];

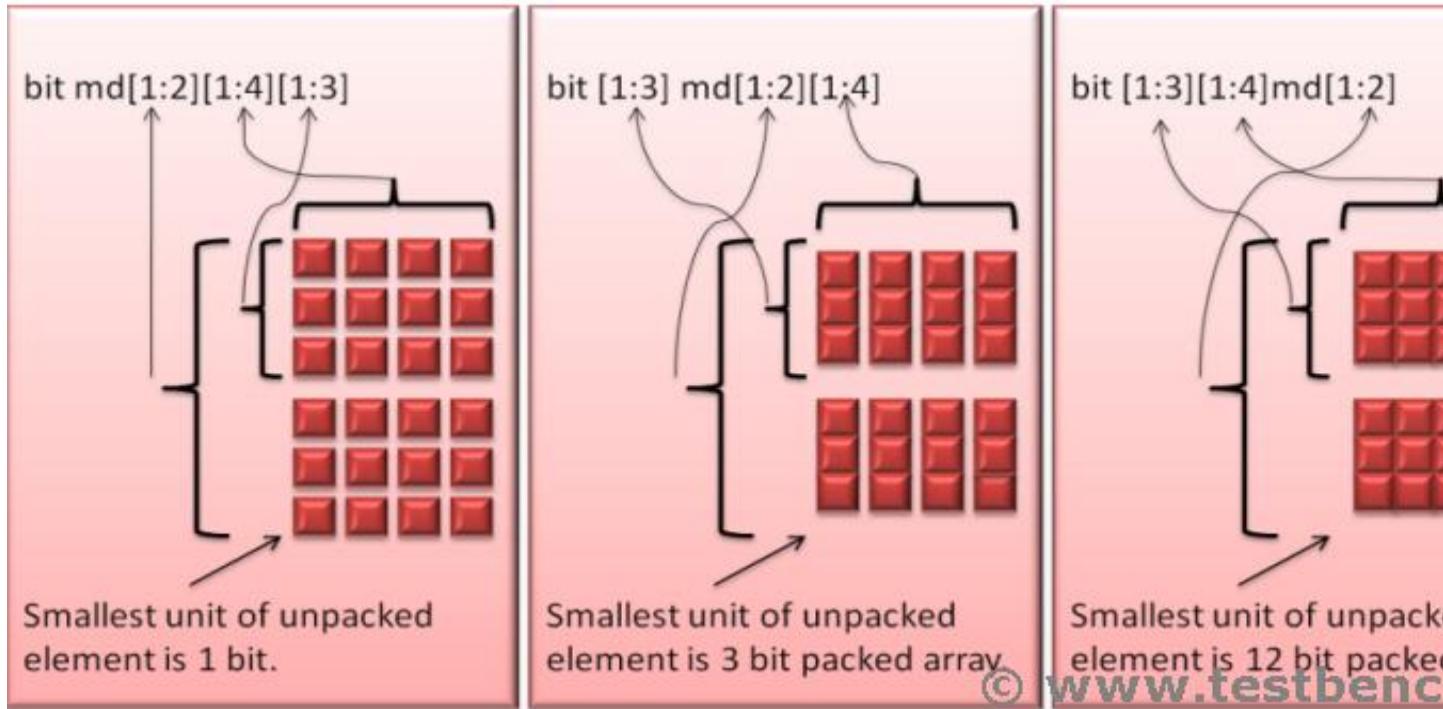


```
// Packed Arrays
reg [0:10] vari; // packed array of 4-bits
wire [31:0] [1:0] vari; // 2-dimensional packed array

// Unpacked Arrays
wire status [31:0]; // 1 dimensional unpacked array
wire status [32]; // 1 dimensional unpacked array

integer matrix[7:0][0:31][15:0]; // 3-dimensional unpacked array of integers
integer matrix[8][32][16]; // 3-dimensional unpacked array of integers

reg [31:0] registers1 [0:255]; // unpacked array of 256 registers; each
reg [31:0] registers2 [256]; // register is packed 32 bit wide
```



## Operations On Arrays

The following operations can be performed on all arrays, packed or unpacked:

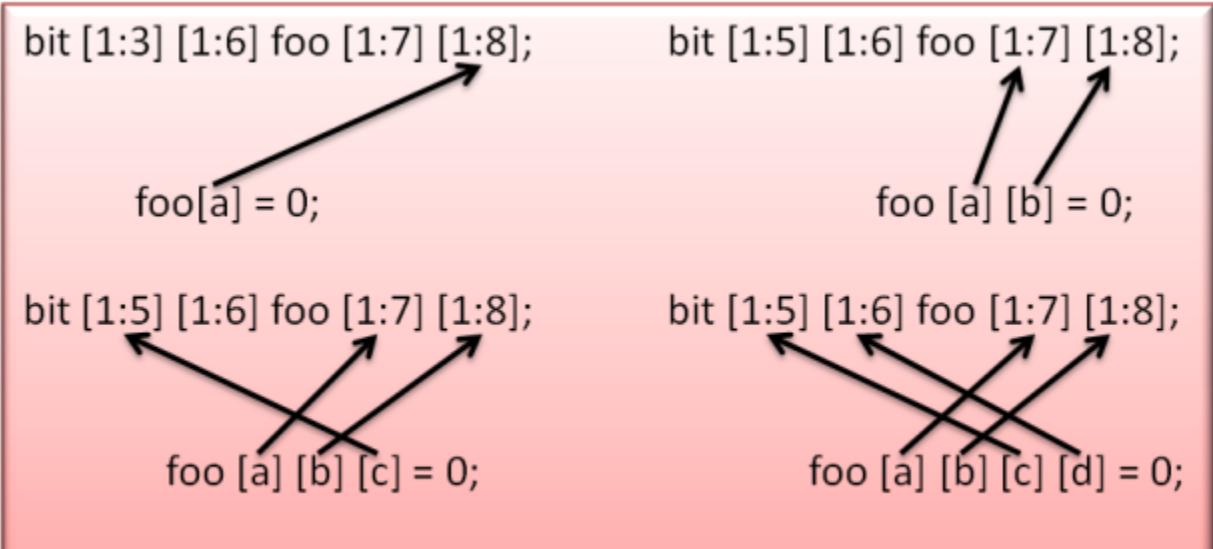
```

register1 [6][7:0] = `1;      // Packed indexes can be sliced
register1 = ~register1 ;     // Can operate on entire memory
register1 = register2 ;      // Reading and writing the entire array
register1[7:4] = register2[3:0] // Reading and writing a slice of the array
register1[4+:i]= register2[4+i]; // Reading and writing a variable slice
if(register1 == register2)   //Equality operations on the entire array
int n[1:2][1:3] = `{{0,1,2},{4,4,3}};// multidimensional assignment
int triple [1:3] = `{{1:1, default:0}}; // indexes 2 and 3 assigned 0

```

## Accessing Individual Elements Of Multidimensional Arrays:

In a list of multi dimensions, the rightmost one varies most rapidly than the left most one. Packed dimension varies more rapidly than an unpacked.



© www.testbench.in

In the following example, each dimension is having unique range and reading and writing to a element shows exactly which index corresponds to which dimension.

```
module index();
bit [1:5][10:16] foo [21:27][31:38];
initial
begin
foo[24][34][4][14] = 1;
$display(" foo[24][34][4][14] is %d ",foo[24][34][4][14]);
end
endmodule
```

The result of reading from an array with an out of the address bounds or if any bit in the address is X or Z shall return the default uninitialized value for the array element type.

As in Verilog, a comma-separated list of array declarations can be made. All arrays in the list shall have the same data type and the same packed array dimensions.

```
module array();
bit [1:5][10:16] foo1 [21:27][31:38],foo2 [31:27][33:38];
initial
begin
$display(" dimensions of foo1 is %d foo2 is %d",$dimensions(foo1),$dimensions(foo2));
$display(" reading with out of bound resulted %d",foo1[100][100][100][100]);
$display(" reading with index x resulted %d",foo1[33][1'bx]);
end
```

```
end  
endmodule  
RESULT:
```

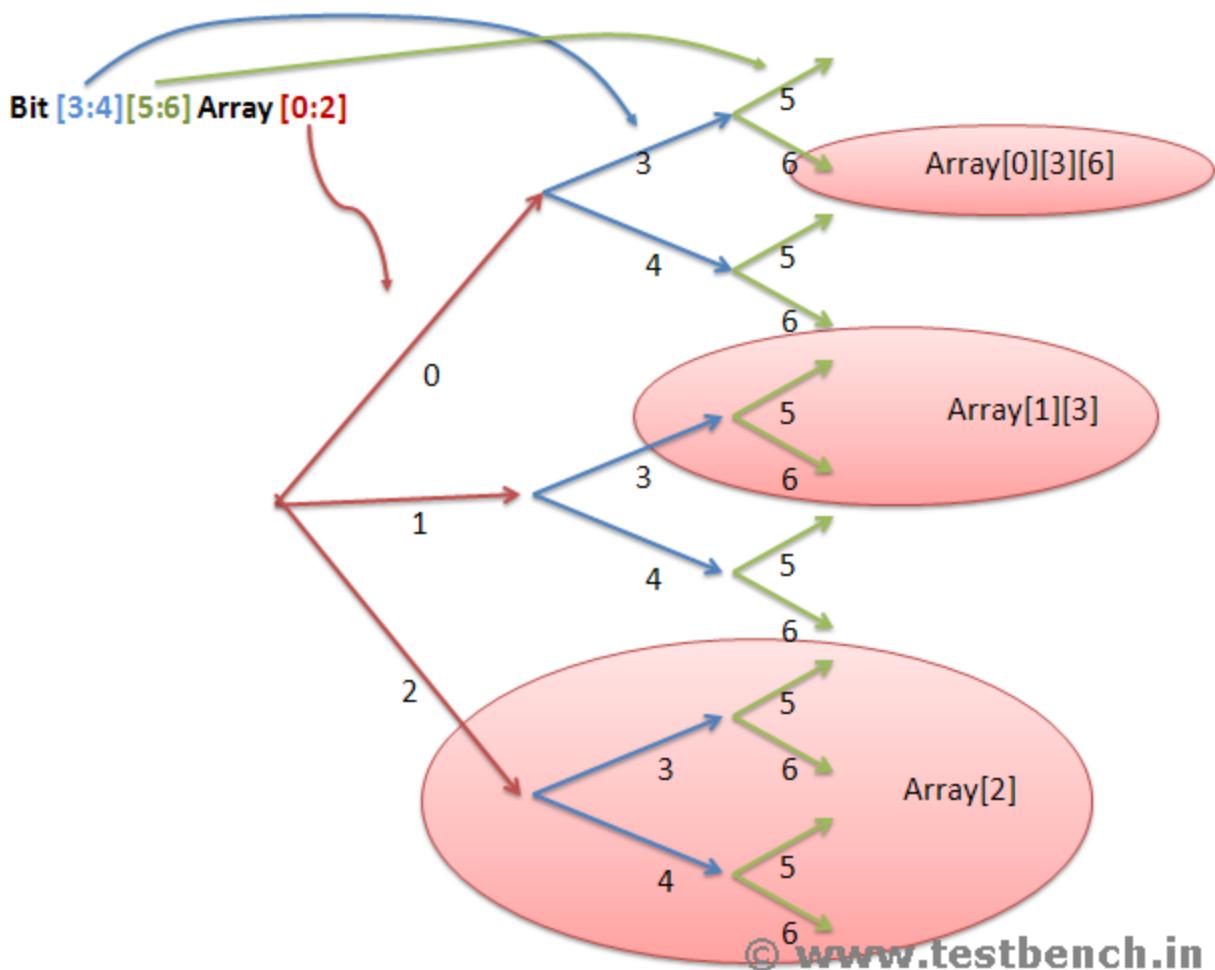
dimensions of foo1 is 4 foo2 is 4  
reading with out of bound resulted x  
reading with index x resulted x

**bit [3:4][5:6]Array [0:2];**

Accessing "Array[2]" will access 4 elements Array[2][3][5],Array[2][3][6],Array[2][4][5] and Array[2][4][6].

Accessing "Array[1][3]" will access 2 elements Array[1][3][5] and Array[1][3][6].

Accessing "Array[0][3][6]" will access one element.



## **ARRAY METHODS**

### **Array Methods:**

SystemVerilog provides various kinds of methods that can be used on arrays. They are

- ⌚ Array querying functions
- ⌚ Array Locator Methods
- ⌚ Array ordering methods
- ⌚ Array reduction methods
- ⌚ Iterator index querying

### **Array Querying Functions:**

SystemVerilog provides new system functions to return information about an array. They are

#### **\$left**

\$left shall return the left bound (MSB) of the dimension.

#### **\$right**

\$right shall return the right bound (LSB) of the dimension.

#### **\$low**

\$low shall return the minimum of \$left and \$right of the dimension.

#### **\$high**

\$high shall return the maximum of \$left and \$right of the dimension.

#### **\$increment**

\$increment shall return 1 if \$left is greater than or equal to \$right and -1 if \$left is less than \$right.

#### **\$size**

\$size shall return the number of elements in the dimension, which is equivalent to \$high - \$low +1.

#### **\$dimensions**

\$dimensions shall return the total number of dimensions in the array.

### **EXAMPLE : arrays**

```
module arr;
bit [2:0][3:0] arr [4:0][5:0];
initial
begin
$display(" $left %0d $right %0d $low %0d $high %0d $increment %0d $size %0d
$dimensions
%0d",$left(arr),$right(arr),$low(arr),$high(arr),$increment(arr),$size(arr),$dimensions(arr));
end
endmodule
```

### **RESULTS :**

\$left 4 \$right 0 \$low 0 \$high 4 \$increment 1 \$size 5 \$dimensions 4

## **Array Locator Methods:**

Array locator methods operate on any unpacked array, including queues, but their return type is a queue. These locator methods allow searching an array for elements (or their indexes) that satisfies a given expression. Array locator methods traverse the array in an unspecified order. The optional "with" expression should not include any side effects; if it does, the results are unpredictable.

The following locator methods are supported (the "with" clause is mandatory) :

### **find()**

find() returns all the elements satisfying the given expression

### **find\_index()**

find\_index() returns the indexes of all the elements satisfying the given expression

### **find\_first()**

find\_first() returns the first element satisfying the given expression

### **find\_first\_index()**

find\_first\_index() returns the index of the first element satisfying the given expression

### **find\_last()**

find\_last() returns the last element satisfying the given expression

### **find\_last\_index()**

find\_last\_index() returns the index of the last element satisfying the given expression

For the following locator methods the "with" clause (and its expression) can be omitted if the relational operators (<, >, ==) are defined for the element type of the given array. If a "with" clause is specified, the relational operators (<, >, ==) must be defined for the type of the expression.

### **min()**

min() returns the element with the minimum value or whose expression evaluates to a minimum

### **max()**

max() returns the element with the maximum value or whose expression evaluates to a maximum

### **unique()**

unique() returns all elements with unique values or whose expression is unique

### **unique\_index()**

unique\_index() returns the indexes of all elements with unique values or whose expression is unique

## **EXAMPLE :**

```
module arr_me;
  string SA[10], qs[$];
  int IA[*], qi[$];
  initial
    begin
      SA[1:5] = { "Bob", "Abc", "Bob", "Henry", "John" };
    end
endmodule
```

```

IA[2]=3;
IA[3]=13;
IA[5]=43;
IA[8]=36;
IA[55]=237;
IA[28]=39;
// Find all items greater than 5
qi = IA.find( x ) with ( x > 5 );
for ( int j = 0; j < qi.size; j++ ) $write("%0d ",qi[j] );
$display("");
// Find indexes of all items equal to 3
qi = IA.find_index with ( item == 3 );
for ( int j = 0; j < qi.size; j++ ) $write("%0d ",qi[j] );
$display("");
// Find first item equal to Bob
qs = SA.find_first with ( item == "Bob" );
for ( int j = 0; j < qs.size; j++ ) $write("%s ",qs[j] );
$display("");
// Find last item equal to Henry
qs = SA.find_last( y ) with ( y == "Henry" );
for ( int j = 0; j < qs.size; j++ ) $write("%s ",qs[j] );
$display("");
// Find index of last item greater than Z
qi = SA.find_last_index( s ) with ( s > "Z" );
for ( int j = 0; j < qi.size; j++ ) $write("%0d ",qi[j] );
$display("");
// Find smallest item
qi = IA.min;
for ( int j = 0; j < qi.size; j++ ) $write("%0d ",qi[j] );
$display("");
// Find string with largest numerical value
qs = SA.max with ( item.atoi );
for ( int j = 0; j < qs.size; j++ ) $write("%s ",qs[j] );
$display("");
// Find all unique strings elements
qs = SA.unique;
for ( int j = 0; j < qs.size; j++ ) $write("%s ",qs[j] );
$display("");
// Find all unique strings in lowercase
qs = SA.unique( s ) with ( s.tolower );

```

```
for ( int j = 0; j < qs.size; j++ ) $write("%s",qs[j] );
end
endmodule
```

## RESULTS :

13\_43\_36\_39\_237\_

2\_

Bob\_

Henry\_

3\_

\_Bob\_Abc\_Henry\_John\_

\_Bob\_Abc\_Henry\_John\_

## Array Ordering Methods:

Array ordering methods can reorder the elements of one-dimensional arrays or queues. The following ordering methods are supported:

### reverse()

reverse() reverses all the elements of the packed or unpacked arrays.

### sort()

sort() sorts the unpacked array in ascending order, optionally using the expression in the with clause.

### rsort()

rsort() sorts the unpacked array in descending order, optionally using the with clause expression.

### shuffle()

shuffle() randomizes the order of the elements in the array.

## EXAMPLE:

```
module arr_order; string s[] = '{ "one", "two", "three" };
initial
begin
    s.reverse;
    for ( int j = 0; j < 3;j++ ) $write("%s",s[j] );
    s.sort;
    for ( int j = 0; j < 3;j++ ) $write("%s",s[j] );
    s.rsort;
    for ( int j = 0; j < 3;j++ ) $write("%s",s[j] );
    s.shuffle;
    for ( int j = 0; j < 3;j++ ) $write("%s",s[j] );
end
endmodule
```

## **RESULT:**

three two one

one three two  
two three one  
three one two

### **Array Reduction Methods :**

Array reduction methods can be applied to any unpacked array to reduce the array to a single value. The expression within the optional "with" clause can be used to specify the item to use in the reduction. The following reduction methods are supported:

#### **sum()**

sum() returns the sum of all the array elements.

#### **product()**

product() returns the product of all the array elements

#### **and()**

and() returns the bit-wise AND ( & ) of all the array elements.

#### **or()**

or() returns the bit-wise OR ( | ) of all the array elements

#### **xor()**

xor() returns the logical XOR ( ^ ) of all the array elements.

## **EXAMPLE:**

```
module array_redu();
    byte b[] = { 1, 2, 3, 4 };
    int sum,product,b_xor;
    initial
    begin
        sum = b.sum ; // y becomes 10 => 1 + 2 + 3 + 4
        product = b.product ; // y becomes 24 => 1 * 2 * 3 * 4
        b_xor = b.xor with ( item + 4 ); // y becomes 12 => 5 ^ 6 ^ 7 ^ 8
        $display(" Sum is %0d, product is %0d, xor is %0b ",sum,product,b_xor);
    end
endmodule
```

## **RESULT**

Sum is 10, product is 24, xor is 1100

### **Iterator Index Querying:**

The expressions used by array manipulation methods sometimes need the actual array indexes at each iteration, not just the array element. The index method of an iterator returns the index value of the specified dimension.

```
// find all items equal to their position (index)
q = arr.find with ( item == item.index );
```

```
// find all items in mem that are greater than corresponding item in mem2  
q = mem.find( x ) with ( x > mem2[x.index(1)][x.index(2)] );
```

## **DYNAMIC ARRAYS**

Verilog does not allow changing the dimensions of the array once it is declared. Most of the time in verification, we need arrays whose size varies based on some behavior. For example Ethernet packet varies length from one packet to other packet. In Verilog, for creating Ethernet packet, array with Maximum packet size is declared and only the number of elements which are required for small packets are used and unused elements are waste of memory.

To overcome this deficiency, System Verilog provides Dynamic Array. A dynamic array is an unpacked array whose size can be set or changed at runtime unlike Verilog which needs size at compile time. Dynamic arrays allocate storage for elements at run time along with the option of changing the size.

### **Declaration Of Dynamic Array:**

```
integer dyna_arr_1[],dyn_arr_2[],multi_dime_dyn_arr[][];
```

### **Allocating Elements:**

New[]: The built-in function new allocates the storage and initializes the newly allocated array elements either to their default initial value.

```
dyna_arr_1 = new[10] ;// Allocating 10 elements  
multi_dime_dyn_arr = new[4]; // subarrays remain unsized and uninitialized
```

### **Initializing Dynamic Arrays:**

The size argument need not match the size of the initialization array. When the initialization array's size is greater, it is truncated to match the size argument; when it is smaller, the initialized array is padded with default values to attain the specified size.

```
dyna_arr_2 = new[4]('{4,5,6}); // elements are {4,5,6,0}
```

### **Resizing Dynamic Arrays:**

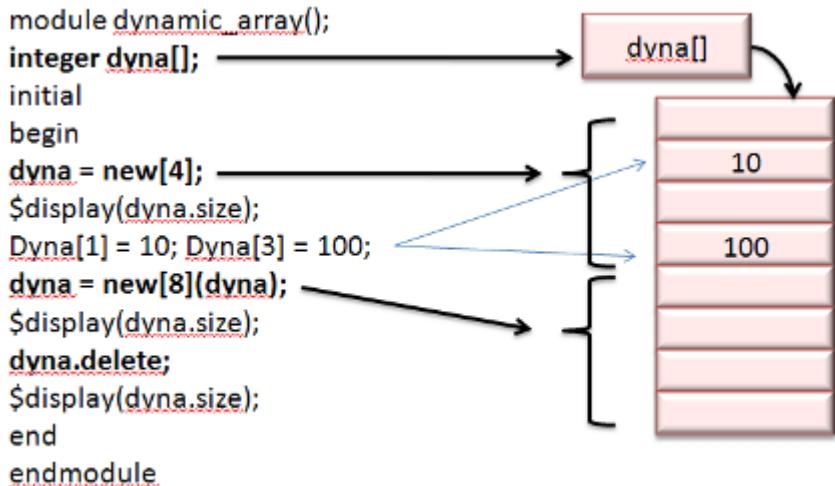
Using new[] constructor and its argument, we can increase the array size without losing the data content.

```
Dyna_arr_1 = new[100] (dyna_arr_1); // Previous 10 data preserved
```

### **Copying Elements:**

Copy constructor of dynamic arrays is an easy and faster way of creating duplicate copies of data.

```
Dyna_arr_2 = new[100](dyna_arr_1); // allocating and copying 100 elements.  
Dyna_arr_1 = [1000]; // Previous data lost. 1000 elements are allocated.
```



© www.testbench.in

## RESULT

```

4
8
0

```

The information about the size of the dynamic array is with the array itself. It can be obtained using `.size()` method. This will be very helpful when you are playing with array. You don't need to pass the size information explicitly. We can also use system task `$size()` method instead of `.size()` method. SystemVerilog also provides `delete()` method clears all the elements yielding an empty array (zero size).

## ASSOCIATIVE ARRAYS

Dynamic arrays are useful for dealing with contiguous collections of variables whose number changes dynamically. Associative arrays give you another way to store information. When the size of the collection is unknown or the data space is sparse, an associative array is a better option. In Associative arrays Elements Not Allocated until Used. Index Can Be of Any Packed Type, String or Class. Associative elements are stored in an order that ensures fastest access. In an associative array a key is associated with a value. If you wanted to store the information of various transactions in an array, a numerically indexed array would not be the best choice. Instead, we could use the transaction names as the keys in associative array, and the value would be their respective information. Using associative arrays, you can call the array element you need using a string rather than a number, which is often easier to remember.

The syntax to declare an associative array is:

```
data_type array_id [ key _type];
```

data\_type is the data type of the array elements.  
array\_id is the name of the array being declared.  
key\_type is the data-type to be used as an key.

Examples of associative array declarations are:

```
int array_name[*];//Wildcard index. can be indexed by any integral datatype.  
int array_name [ string ];// String index  
int array_name [ some_Class ];// Class index  
int array_name [ integer ];// Integer index  
typedef bit signed [4:1] Nibble;  
int array_name [ Nibble ]; // Signed packed array
```

Elements in associative array elements can be accessed like those of one dimensional arrays.  
Associative array literals use the '{index:value}' syntax with an optional default index.

//associative array of 4-state integers indexed by strings, default is '1.

```
integer tab [string] = {'Peter':20, "Paul":22, "Mary":23, default:-1 };
```

### Associative Array Methods

SystemVerilog provides several methods which allow analyzing and manipulating associative arrays. They are:

The num() or size() method returns the number of entries in the associative array.

The delete() method removes the entry at the specified index.

The exists() function checks whether an element exists at the specified index within the given array.

The first() method assigns to the given index variable the value of the first (smallest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.

The last() method assigns to the given index variable the value of the last (largest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.

The next() method finds the entry whose index is greater than the given index. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1.

Otherwise, the index is unchanged, and the function returns 0.

The prev() function finds the entry whose index is smaller than the given index. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.

### **EXAMPLE**

```
module assoc_arr;  
    int temp,imem[*];  
    initial  
    begin
```

```

imem[ 2'd3 ] = 1;
imem[ 16'hfff ] = 2;
imem[ 4'b1000 ] = 3;
$display( "%0d entries", imem.num );
if(imem.exists( 4'b1000 ) )
    $display("imem.exists( 4b'1000 )");
imem.delete(4'b1000);
if(imem.exists( 4'b1000 ) )
    $display(" imem.exists( 4b'1000 )");
else
    $display(" ( 4b'1000) not existing");
if(imem.first(temp))
    $display(" First entry is at index %0db ",temp);
if(imem.next(temp))
    $display(" Next entry is at index %0h after the index 3",temp);
// To print all the elements alone with its indexes
if (imem.first(temp) )
do
    $display( "%d : %d", temp, imem[temp] );
while (imem.next(temp));
end
endmodule

```

## RESULT

3 entries  
imem.exists( 4b'1000)  
( 4b'1000) not existing  
First entry is at index 3b  
Next entry is at index ffff after the index 3  
3 : 1  
65535 : 2

## QUEUES

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. Queues support insertion and deletion of elements from random locations using an index. Queues Can be passed to tasks / functions as ref or non-ref arguments. Type checking is also done.

## Queue Operators:

Queues and dynamic arrays have the same assignment and argument passing semantics. Also, queues support the same operations that can be performed on unpacked arrays and use the same operators and rules except as defined below:

```
int q[$] = { 2, 4, 8 };
int p[$];
int e, pos;
e = q[0];           // read the first (leftmost) item
e = q[$];          // read the last (rightmost) item
q[0] = e;          // write the first item
p = q;              // read and write entire queue (copy)
q = { q, 6 };       // insert '6' at the end (append 6)
q = { e, q };       // insert 'e' at the beginning (prepend e)
q = q[1:$];         // delete the first (leftmost) item
q = q[0:$-1];       // delete the last (rightmost) item
q = q[1:$-1];       // delete the first and last items
q = {};              // clear the queue (delete all items)
q = { q[0:pos-1], e, q[pos,$] }; // insert 'e' at position pos
q = { q[0:pos], e, q[pos+1,$] }; // insert 'e' after position pos
```

## Queue Methods:

In addition to the array operators, queues provide several built-in methods. They are:

The size() method returns the number of items in the queue. If the queue is empty, it returns 0.

The insert() method inserts the given item at the specified index position.

The delete() method deletes the item at the specified index.

The pop\_front() method removes and returns the first element of the queue.

The pop\_back() method removes and returns the last element of the queue.

The push\_front() method inserts the given element at the front of the queue.

The push\_back() method inserts the given element at the end of the queue.

## EXAMPLE

```
module queues;
byte qu[$];
initial
begin
qu.push_front(2);
```

```

qu.push_front(12);
qu.push_front(22);
qu.push_back(11);
qu.push_back(99);
$display(" %d ",qu.size() );
$display(" %d ",qu.pop_front() );
$display(" %d ",qu.pop_back() );
qu.delete(3);
$display(" %d ",qu.size() );
end
endmodule

```

#### **RESULTS :**

```

5
22
99

```

#### **Dynamic Array Of Queues Queues Of Queues**

##### **EXAMPLE:**

```

module top;
  typedef int qint_t[$];
  // dynamic array of queues
  qint_t DAq[]; // same as int DAq[][][];
  // queue of queues
  qint_t Qq[$]; // same as int Qq[$][$];
  // associative array of queues
  qint_t AAq[string]; // same as int AAq[string][];
initial begin
  // Dynamic array of 4 queues
  DAq = new[4];
  // Push something onto one of the queues
  DAq[2].push_back(1);
  // initialize another queue with three entries
  DAq[0] = {1,2,3};
  $display("%p",DAq);
  // Queue of queues -two
  Qq= {{1,2},{3,4,5}};
  Qq.push_back(qint_t'{6,7});
  Qq[2].push_back(1);
  $display("%p",Qq);
  // Associative array of queues
  AAq["one"] = {};

```

```

AAq["two"] = {1,2,3,4};
AAq["one"].push_back(5);
$display("%p",AAq);
end
endmodule : top

```

## **RESULTS:**

```

'{{1, 2, 3}, {}, {1}, {}}
'{{1, 2}, {3, 4, 5}, {6, 7, 1}}
'{one:{5}, two:{1, 2, 3, 4} }

```

## **COMPARISON OF ARRAYS**

### **Static Array**

Size should be known at compilation time.

Time require to access any element is less.

if not all elements used by the application, then memory is wasted.

Not good for sparse memory or when the size changes.

Good for contagious data.

### **Associativearray**

No need of size information at compile time.

Time require to access an element increases with size of the array.

Compact memory usage for sparse arrays.

User don't need to keep track of size. It is automatically resized.

Good inbuilt methods for Manipulating and analyzing the content.

### **Dynamicarray**

No need of size information at compile time.

To set the size or resize, the size should be provided at runtime.

Performance to access elements is same as Static arrays.

Good for contagious data.

Memory usage is very good, as the size can be changed dynamically.

### **Queues**

No need of size information at compile time.

Performance to access elements is same as Static arrays.

User doesn't need to provide size information to change the size. It is automatically resized.

Rich set of inbuilt methods for Manipulating and analyzing the content.

Useful in self-checking modules. Very easy to work with out of order transactions.

Inbuilt methods for sum of elements, sorting all the elements.

Searching for elements is very easy even with complex expressions.

Useful to model FIFO or LIFO.

Array type	Memory	Performance	Manipulating / Analyzing capability
<b>Fixed</b>	Ok	Good	Ok
<b>Associative</b>	Good	Ok	Ok
<b>Dynamic</b>	Good	Good	Ok
<b>Queue</b>	Good	Good	Very Good

© www.testbench.in

## LINKED LIST

The List package implements a classic list data-structure, and is analogous to the STL (Standard Template Library) List container that is popular with C++ programmers. The container is defined as a parameterized class, meaning that it can be customized to hold data of any type. The List package supports lists of any arbitrary predefined type, such as integer, string, or class object. First declare the Linked list type and then take instances of it. SystemVerilog has many methods to operate on these instances.

A double linked list is a chain of data structures called nodes. Each node has 3 members, one points to the next item or points to a null value if it is last node, one points to the previous item or points to a null value if it is first node and other has the data.



© www.testbench.in

The disadvantage of the linked list is that data can only be accessed sequentially and not in random order. To read the 1000th element of a linked list, you must read the 999 elements that precede it.

### List Definitions:

list :- A list is a doubly linked list, where every element has a predecessor and successor. It is a sequence that supports both forward and backward traversal, as well as amortized constant time insertion and removal of elements at the beginning, end, or middle.

container :- A container is a collection of objects of the same type .Containers are objects that contain and manage other objects and provide iterators that allow the contained objects (elements) to be addressed. A container has methods for accessing its elements. Every container

has an associated iterator type that can be used to iterate through the container's elements.  
iterator :- Iterators provide the interface to containers. They also provide a means to traverse the container elements. Iterators are pointers to nodes within a list. If an iterator points to an object in a range of objects and the iterator is incremented, the iterator then points to the next object in the range.

### **Procedure To Create And Use List:**

1. include the generic List class declaration  
``include <List.vh>`
2. Declare list variable  
`List#(integer) il; // Object il is a list of integer`
3. Declaring list iterator  
`List_Iterator#(integer) itor; //Object s is a list-of-integer iterator`

### **List iterator Methods**

The List\_Iterator class provides methods to iterate over the elements of lists.

The next() method changes the iterator so that it refers to the next element in the list.

The prev() method changes the iterator so that it refers to the previous element in the list.

The eq() method compares two iterators and returns 1 if both iterators refer to the same list element.

The neq() method is the negation of eq().

The data() method returns the data stored in the element at the given iterator location.

### **List Methods**

The List class provides methods to query the size of the list; obtain iterators to the head or tail of the list;

retrieve the data stored in the list; and methods to add, remove, and reorder the elements of the list.

The size() method returns the number of elements stored in the list.

The empty() method returns 1 if the number elements stored in the list is zero and 0 otherwise.

The push\_front() method inserts the specified value at the front of the list.

The push\_back() method inserts the specified value at the end of the list.

The front() method returns the data stored in the first element of the list.

The back() method returns the data stored in the last element of the list.

The pop\_front() method removes the first element of the list.

The pop\_back() method removes the last element of the list.

The start() method returns an iterator to the position of the first element in the list.

The finish() method returns an iterator to a position just past the last element in the list.

The insert() method inserts the given data into the list at the position specified by the iterator.

The insert\_range() method inserts the elements contained in the list range specified by the

iterators first and last at the specified list position.

The `erase()` method removes from the list the element at the specified position.

The `erase_range()` method removes from a list the range of elements specified by the first and last iterators.

The `set()` method assigns to the list object the elements that lie in the range specified by the first and last iterators.

The `swap()` method exchanges the contents of two equal-size lists.

The `clear()` method removes all the elements from a list, but not the list itself.

The `purge()` method removes all the list elements (as in `clear`) and the list itself.

## EXAMPLE

```
module lists();
List#(integer) List1;
List_Iterator#(integer) itor;
initial begin
    List1 = new();
    $display (" size of list is %d \n",List1.size());
    List1.push_back(10);
    List1.push_front(22);
    $display (" size of list is %d \n",List1.size());
    $display (" poping from list : %d \n",List1.front());
    $display (" poping from list : %d \n",List1.front());
    List1.pop_front();
    List1.pop_front();
    $display (" size of list is %d \n",List1.size());
    List1.push_back(5);
    List1.push_back(55);
    List1.push_back(555);
    List1.push_back(5555);
    $display (" size of list is %d \n",List1.size());

    itor = List1.start();
    $display (" startn of list %d \n",itor.data());
    itor.next();
    $display (" second element of list is %d \n",itor.data());
    itor.next();
    $display (" third element of list is %d \n",itor.data());
    itor.next();
    $display (" fourth element of list is %d \n",itor.data());

    itor = List1.erase(itor);
```

```

$display (" after erasing element,the itor element of list is %d \n",itor.data());
itor.prev();
$display(" prevoious element is %d \n",itor.data());
end
endmodule

```

### **RESULT:**

```

size of list is 0
size of list is 2
poping from list : 22
poping from list : 22
size of list is 0
size of list is 4
startn of list 5
second element of list is 55
third element of list is 555
fourth element of list is 5555
after erasing element,the itor element of list is x
prevoious element is 555

```

## **CASTING**

Verilog is loosely typed . Assignments can be done from one data type to other data types based on predefined rules. The compiler only checks that the destination variable and source expression are scalars. Otherwise, no type checking is done at compile time. Systemverilog has complex data types than Verilog. It's necessary for SystemVerilog to be much stricter about type conversions than Verilog, so Systemverilog provided the cast(`) operator, which specifies the type to use for a specific expression. Using Casting one can assign values to variables that might not ordinarily be valid because of differing data type. SystemVerilog adds 2 types of casting. Static casting and dynamic casting.

### **Static Casting**

A data type can be changed by using a cast ( ` ) operation. In a static cast, the expression to be cast shall be enclosed in parentheses that are prefixed with the casting type and an apostrophe. If the expression is assignment compatible with the casting type, then the cast shall return the value that a variable of the casting type would hold after being assigned the expression.

### **EXAMPLE:**

```

int'(2.0 * 3.0)
shortint'({8'hFA,8'hCE})
signed'(x)
17'(x - 2)

```

## Dynamic Casting

SystemVerilog provides the \$cast system task to assign values to variables that might not ordinarily be valid because of differing data type. \$cast can be called as either a task or a function.

The syntax for \$cast is as follows:

```
function int $cast( singular dest_var, singular source_exp );
```

or

```
task $cast( singular dest_var, singular source_exp );
```

The dest\_var is the variable to which the assignment is made. The source\_exp is the expression that is to be assigned to the destination variable. Use of \$cast as either a task or a function determines how invalid assignments are handled. When called as a task, \$cast attempts to assign the source expression to the destination variable. If the assignment is invalid, a run-time error occurs, and the destination variable is left unchanged.

**EXAMPLE:**

```
typedef enum { red, green, blue, yellow, white, black } Colors;  
Colors col;  
$cast( col, 2 + 3 );
```

## Cast Errors

Following example shows the compilation error.

**EXAMPLE:**

```
module enum_method;  
  typedef enum {red,blue,green} colour;  
  colour c,d;  
  int i;  
  initial  
  begin  
    d = (c + 1);  
  end  
endmodule
```

## RESULT

Illegal assignment

Following example shows the simulation error. This is compilation error free. In this example , d is assigned c + 10 , which is out of bound in enum colour.

**EXAMPLE:**

```
module enum_method;  
  typedef enum {red,blue,green} colour;  
  colour c,d;  
  int i;  
  initial
```

```

begin
    $cast(d,c + 10);
end
endmodule

```

## **RESULT**

Dynamic cast failed

## **DATA DECLARATION**

### **Scope And Lifetime:**

#### **Global :**

SystemVerilog adds the concept of global scope. Any declarations and definitions which is declared outside a module, interface, task, or function, is global in scope. Global variables have a static lifetime (exists for the whole elaboration and simulation time). Datatypes, tasks, functions, class definitions can be in global scope. Global members can be referenced explicitly via the \$root . All these can be accessed from any scope as this is the highest scope and any other scope will be below the global.

#### **Local :**

Local declarations and definitions are accessible at the scope where they are defined and below. By default they are static in life time. They can be made to automatic. To access these local variables which are static, hierarchical pathname should be used.

```

int st0;          //Static. Global Variable. Declared outside module
task disp();      //Static. Global Task.
module msl;
    int st0;          //static. Local to module
    initial begin
        int st1;          //static. Local to module
        static int st2;    //static. Local to Module
        automatic int auto1; //automatic.
    end
    task automatic t1(); //Local task definition.
        int auto2;        //automatic. Local to task
        static int st3;    //static.Local to task. Hierarchical path access allowed
        automatic int auto3; //automatic. Local to task
        $root.st0 = st0;   // $root.st0 is global variable, st0 is local to module.
    endtask
endmodule

```

#### **Alias:**

The Verilog assign statement is a unidirectional assignment. To model a bidirectional short-circuit connection it is necessary to use the alias statement.

This example strips out the least and most significant bytes from a four byte bus:

```

module byte_rip (inout wire [31:0] W, inout wire [7:0] LSB, MSB);
  alias W[7:0] = LSB;
  alias W[31:24] = MSB;
endmodule

```

### Data Types On Ports:

Verilog restricts the data types that can be connected to module ports. Only net types are allowed on the receiving side and Nets, regs or integers on the driving side. SystemVerilog removes all restrictions on port connections. Any data type can be used on either side of the port. Real numbers, Arrays, Structures can also be passed through ports.

### Parameterized Data Types:

Verilog allowed only values to be parameterized. SystemVerilog allows data types to be "parameterized". A data-type parameter can only be set to a data-type.

```

module foo #(parameter type VAR_TYPE = integer);
  foo #(.VAR_TYPE(byte)) bar ();

```

### Declaration And Initialization:

```

integer i = 1;

```

In Verilog, an initialization value specified as part of the declaration is executed as if the assignment were made from an initial block, after simulation has started. This creates an event at time 0 and it is same as if the assignment is done in initial block. In Systemverilog , setting the initial value of a static variable as part of the variable declaration is done before initial block and so does not generate an event.

### REG AND LOGIC

Historically, Verilog used the terms wire and reg as a descriptive way to declare wires and registers. The original intent was soon lost in synthesis and verification coding styles, which soon gave way to using terms Nets and Variables in Verilog-2001. The keyword reg remained in SystemVerilog, but was now misleading its intent. SystemVerilog adds the keyword logic as a more descriptive term to remind users that it is not a hardware register. logic and reg are equivalent types.

SystemVerilog extended the variable type so that, it can be used to connect gates and modules. All variables can be written either by one continuous assignment, or by one or more procedural statements. It shall be an error to have multiple continuous assignments or a mixture of procedural and continuous assignments.

Now we saw logic and wire are closer. Wire (net) is used when driven by multiple drivers, where as logic is used when it is driven by only one driver. logic just holds the last value assigned to it, while a wire resolves its value based on all the drivers.

For example:

```

logic abc;

```

The following statements are legal assignments to logic abc:

- 1) **assign** abc = sel ? 1 : 0;
- 2) **not** (abc,pqr),
- 3) **always #10** abc = ~abc;

## **OPERATORS 1**

The SystemVerilog operators are a combination of Verilog and C operators. In both languages, the type and size of the operands is fixed, and hence the operator is of a fixed type and size. The fixed type and size of operators is preserved in SystemVerilog. This allows efficient code generation.

Verilog does not have assignment operators or increment and decrement operators.

SystemVerilog includes the C assignment operators, such as `+=`, and the C increment and decrement operators, `++` and `--`.

Verilog-2001 added signed nets and reg variables, and signed based literals. There is a difference in the rules for combining signed and unsigned integers between Verilog and C. SystemVerilog uses the Verilog rules.

### **Operators In Systemverilog**

Following are the operators in systemverilog

Operator token	Name	Operand data types
=	binary assignment operator	any
+= -= /= *=	binary arithmetic assignment operators	integral, real, shortreal
%=	binary arithmetic modulus assignment operator	integral
&=  = ^=	binary bit-wise assignment operators	integral
>>= <<=	binary logical shift assignment operators	integral
>>>= <<<=	binary arithmetic shift assignment operators	integral
?:	conditional operator	any
+ -	unary arithmetic operators	integral, real, shortreal
!	unary logical negation operator	integral, real, shortreal
~ & ~&   ~  ~~ ^~	unary logical reduction operators	integral
+ - * / **	binary arithmetic operators	integral, real, shortreal
%	binary arithmetic modulus operator	integral
&   ^ ^~ ^~	binary bit-wise operators	integral
>> <<	binary logical shift operators	integral
>>> <<<	binary arithmetic shift operators	integral
&&    -> <->	binary logical operators	integral, real, shortreal
< <= > >=	binary relational operators	integral, real, shortreal
==!=	binary case equality operators	any except real and shortreal
== !=	binary logical equality operators	any
==? !=?	binary wildcard equality operators	integral
++ --	unary increment, decrement operators	integral, real, shortreal
inside	binary set membership operator	singular for the left operand
dist	binary distribution operator	integral
{ } { { } }	concatenation, replication operators	integral
{<<{ } } {>>{ } }	stream operators	integral

© www.testbench.in

## Assignment Operators

In addition to the simple assignment operator, =, SystemVerilog includes the C assignment operators and special bitwise assignment operators:

+=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=.

An assignment operator is semantically equivalent to a blocking assignment, with the exception that any left-hand side index expression is only evaluated once.

For example:

```
a[i]+=2; // same as a[i] = a[i] +2;
```

Following are the new SystemVerilog assignment operators and its equivalent in verilog

SystemVerilog	Verilog
A += B	A = A + B
A -= B	A = A - B
A *= B	A = A * B
A /= B	A = A / B
A %= B	A = A % B
A &= B	A = A & B
A  = B	A = A   B
A ^= B	A = A ^ B
A <<= B	A = A << B
A >>= B	A = A >> B
A <<<= B	A = A <<< B
A >>>= B	A = A >>> B
A = B++	A = B; B = B + 1;
A = ++B	B = B + 1; A = B;
A = B--	A = B; B = B -1;
A = --A	B = B - 1; A = B;

© www.testbench.in

## Assignments In Expression

In SystemVerilog, an expression can include a blocking assignment. such an assignment must be enclosed in parentheses to avoid common mistakes such as using a=b for a==b, or a|=b for a!=b.

```
if ((a=b)) b = (a+=1); // assign b to a
```

```
a = (b = (c = 5));// assign 5 to c
```

```
if(a=b) // error in systemverilog
```

(a=b) statement assigns b value to a and then returns a value.

if((a=b)) is equivalent to

```
a=b;
```

```
if(a)
```

## EXAMPLE

```
module assignment();
```

```

int a,b,c;
initial begin
    a = 1; b = 2;c = 3;
    if((a==b))
        $display(" a value is %d ",a);
    a = (b = (c = 5));
    $display(" a is %d b is %d c is %d ",a,b,c);
end
endmodule

```

## **RESULT**

a value is 2  
 a is 5 b is 5 c is 5

## **Concatenation :**

{ } concatenation right of assignment.  
 ,{} concatenation left of assignment.

## **EXAMPLE :Concatenation**

```

program main ;
    bit [4:0] a;
    reg b,c,d;
    initial begin
        b = 0;
        c = 1;
        d = 1;
        a = {b,c,0,0,d};
        {b,c,d} = 3'b111;
        $display(" a %b b %b c %b d %b ",a,b,c,d);
    end
endprogram

```

## **RESULTS**

a 00001 b 1 c 1 d 1

## **Arithmetic:**

Operator Token	Operator name
-	unary arithmetic (negative)
+, -, *, /	binary arithmetic
%	modulus
**	power operator

## EXAMPLE :Arithmetic

```
program main;
    integer a,b;
    initial
    begin
        b = 10;
        a = 22;
        $display(" -(nagation) is %0d ",-(a));
        $display(" a + b is %0d ",a+b);
        $display(" a - b is %0d ",a-b);
        $display(" a * b is %0d ",a*b);
        $display(" a / b is %0d ",a/b);
        $display(" a modulus b is %0d ",a%b);
    end
endprogram
```

## RESULTS

```
-(nagation) is -22
a + b is 32
a - b is 12
a * b is 220
a / b is 2
a modules b is 2
```

Following tabel shows the opwer operator rules for calculating the result.

	op1 is negative < -1	op1 is -1	op1 is zero	op1 is 1	op1 is positive > 1
op2 is positive odd	op1 ** op2	-1	0	1	op1 ** op2
op2 is positive even	op1 ** op2	1	0	1	op1 ** op2
op2 is zero	1	1	1	1	1
op2 is negative odd	0	-1	x	1	0
op2 is negative even	0	1	x	1	0

© www.testbench.in

```
program main;
    integer op1_neg,op1_n1,op1_0,op1_p1,op1_pos;
    integer op2_pos_odd,op2_pos_even,op2_zero,op2_neg_odd,op2_neg_even;
    initial
    begin
```

```

op1_neg = -10;op1_n1 = -1;op1_0 = 0;op1_p1 = 1;op1_pos = 10;
op2_pos_odd = 9;op2_pos_even = 10;op2_zero=0;op2_neg_odd =-9;op2_neg_even=-10;
$display(" | -10 -1 0 1 10");
$display(" ---+-----");
$display(" 9| %d %d %d %d
%d",op1_neg**op2_pos_odd,op1_n1**op2_pos_odd,op1_0**op2_pos_odd,op1_p1**op2_pos_
odd,op1_pos**op2_pos_odd);
$display(" 10| %d %d %d %d
%d",op1_neg**op2_pos_even,op1_n1**op2_pos_even,op1_0**op2_pos_even,op1_p1**op2_p
os_even,op1_pos**op2_pos_even);
$display(" 0| %d %d %d %d
%d",op1_neg**op2_zero,op1_n1**op2_zero,op1_0**op2_zero,op1_p1**op2_zero,op1_pos**o
p2_zero);
$display(" -9| %d %d %d %d
%d",op1_neg**op2_neg_odd,op1_n1**op2_neg_odd,op1_0**op2_neg_odd,op1_p1**op2_neg_
odd,op1_pos**op2_neg_odd);
$display("-10| %d %d %d %d
%d",op1_neg**op2_neg_even,op1_n1**op2_neg_even,op1_0**op2_neg_even,op1_p1**op2_n
eg_even,op1_pos**op2_neg_even);
end
endprogram

```

## RESULT

	-10	-1	0	1	10	
	---					
9	3294967296	4294967295		0	1	10000000000
10	1410065408		1	0	1	1410065408
0	1	1	1	1	1	
-9	0	4294967295	x	1	0	
-10	0	1	x	1	0	

## Relational:

# > >= < <= relational

## EXAMPLE :Relational

```

program main ;
  integer a,b;
  initial
  begin
    b = 10;
    a = 22;
  
```

```

$display(" a < b is %0d \n",a < b);
$display(" a > b is %0d \n",a > b);
$display(" a <= b is %0d \n",a <= b);
$display(" a >= b is %0d \n",a >= b);

end
endprogram

```

**RESULTS**

```

a < b is 0
a > b is 1
a <= b is 0
a >= b is 1

```

**Equality:**

Operator Token	Operator name	Definition
a == b	Logical equality	a equal to b, result can be unknown
a != b	Logical inequality	a not equal to b, result can be unknown
a === b	Case equality	a equal to b, including x and z
a !== b	Case inequality	a not equal to b, including x and z
a ==? b	Wildcard equality	a equals b, X and Z values in b act as wildcards
a !=? b	Wildcard inequality	a does not equal b, X and Z values in b act as wildcards

© www.testbench.in

The different types of equality (and inequality) operators in SystemVerilog behave differently when their operands contain unknown values (X or Z). The == and != operators may result in X if any of their operands contains an X or Z. The === and !== check the 4-state explicitly, therefore, X and Z values shall either match or mismatch, never resulting in X. The ==? and !=? operators may result in X if the left operand contains an X or Z that is not being compared with a wildcard in the right operand.

#### EXAMPLE : logical Equality

```

program main;
  reg[3:0] a;
  reg[7:0] x, y, z;
initial begin
  a = 4'b0101;
  x = 8'b1000_0101;
  y = 8'b0000_0101;

```

```

z = 8'b0xx0_0101;
if (x == a)
    $display("x equals a is TRUE.\n");
if (y == a)
    $display("y equals a is TRUE.\n");
if (z == a)
    $display("z equals a is TRUE.\n");
end
endprogram

```

### RESULTS:

y equals a is TRUE.

### EXAMPLE:case equality:

```

program main ;
    reg a_1,a_0,a_x,a_z;
    reg b_1,b_0,b_x,b_z;
    initial
    begin
        a_1 = 'b1;a_0 = 'b0;a_x = 'bx;a_z = 'bz;
        b_1 = 'b1;b_0 = 'b0;b_x = 'bx;b_z = 'bz;
        $display("-----");
        $display (" == 0 1 x z ");
        $display("-----");
        $display (" 0  %b  %b  %b ",a_0 == b_0,a_0 == b_1,a_0 == b_x,a_0 == b_z);
        $display (" 1  %b  %b  %b ",a_1 == b_0,a_1 == b_1,a_1 == b_x,a_1 == b_z);
        $display (" x  %b  %b  %b ",a_x == b_0,a_x == b_1,a_x == b_x,a_x == b_z);
        $display (" z  %b  %b  %b ",a_z == b_0,a_z == b_1,a_z == b_x,a_z == b_z);
        $display("-----");
        $display("-----");
        $display (" === 0 1 x z ");
        $display("-----");
        $display (" 0  %b  %b  %b ",a_0 === b_0,a_0 === b_1,a_0 === b_x,a_0 === b_z);
        $display (" 1  %b  %b  %b ",a_1 === b_0,a_1 === b_1,a_1 === b_x,a_1 === b_z);
        $display (" x  %b  %b  %b ",a_x === b_0,a_x === b_1,a_x === b_x,a_x === b_z);
        $display (" z  %b  %b  %b ",a_z === b_0,a_z === b_1,a_z === b_x,a_z === b_z);
        $display("-----");
        $display("-----");
        $display (" =?= 0 1 x z ");
        $display("-----");
        $display (" 0  %b  %b  %b ",a_0 =?= b_0,a_0 =?= b_1,a_0 =?= b_x,a_0 =?= b_z);

```

```

$display (" 1  %b  %b  %b  %b ",a_1 == b_0,a_1 == b_1,a_1 == b_x,a_1 == b_z);
$display (" x  %b  %b  %b  %b ",a_x == b_0,a_x == b_1,a_x == b_x,a_x == b_z);
$display (" z  %b  %b  %b  %b ",a_z == b_0,a_z == b_1,a_z == b_x,a_z == b_z);
$display("-----");
$display("-----");
$display (" != 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 != b_0,a_0 != b_1,a_0 != b_x,a_0 != b_z);
$display (" 1  %b  %b  %b  %b ",a_1 != b_0,a_1 != b_1,a_1 != b_x,a_1 != b_z);
$display (" x  %b  %b  %b  %b ",a_x != b_0,a_x != b_1,a_x != b_x,a_x != b_z);
$display (" z  %b  %b  %b  %b ",a_z != b_0,a_z != b_1,a_z != b_x,a_z != b_z);
$display("-----");
$display("-----");
$display (" !== 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 !== b_0,a_0 !== b_1,a_0 !== b_x,a_0 !== b_z);
$display (" 1  %b  %b  %b  %b ",a_1 !== b_0,a_1 !== b_1,a_1 !== b_x,a_1 !== b_z);
$display (" x  %b  %b  %b  %b ",a_x !== b_0,a_x !== b_1,a_x !== b_x,a_x !== b_z);
$display (" z  %b  %b  %b  %b ",a_z !== b_0,a_z !== b_1,a_z !== b_x,a_z !== b_z);
$display("-----");
$display("-----");
$display (" !=? 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 !=? b_0,a_0 !=? b_1,a_0 !=? b_x,a_0 !=? b_z);
$display (" 1  %b  %b  %b  %b ",a_1 !=? b_0,a_1 !=? b_1,a_1 !=? b_x,a_1 !=? b_z);
$display (" x  %b  %b  %b  %b ",a_x !=? b_0,a_x !=? b_1,a_x !=? b_x,a_x !=? b_z);
$display (" z  %b  %b  %b  %b ",a_z !=? b_0,a_z !=? b_1,a_z !=? b_x,a_z !=? b_z);
$display("-----");

```

**end**

**endprogram**

**RESULTS**

---

== 0 1 x z

---

0 1 0 x x  
1 0 1 x x  
x x x x x  
z x x x x

---

=====

==== 0 1 x z

=====

0 1 0 0 0  
1 0 1 0 0  
x 0 0 1 0  
z 0 0 0 1

=====

=====

=?= 0 1 x z

=====

0 1 0 1 1  
1 0 1 1 1  
x 1 1 1 1  
z 1 1 1 1

=====

=====

!= 0 1 x z

=====

0 0 1 x x  
1 1 0 x x  
x x x x x  
z x x x x

=====

=====

!= 0 1 x z

=====

0 0 1 1 1  
1 1 0 1 1  
x 1 1 0 1  
z 1 1 1 0

=====

=====

?= 0 1 x z

=====

0 0 1 0 0  
1 1 0 0 0  
x 0 0 0 0  
z 0 0 0 0

=====

## **OPERATORS 2**

### **Logical :**

Operator Token	Operator name
!	Logical negation
&&	Logical and
	Logical or
->	Logical implication
<->	Logical equivalence

© www.testbench.in

SystemVerilog added two new logical operators logical implication (->), and logical equivalence (<->). The logical implication expression1 -> expression2 is logically equivalent to (!expression1 || expression2), and the logical equivalence expression1 <-> expression2 is logically equivalent to ((expression1 -> expression2) && (expression2 -> expression1)).

SystemVerilog	Verilog Equivalent
Exp1-> Exp2	(!Exp1    Exp2)
Exp1<-> Exp2	((Exp1-> Exp2)&&(Exp2-> Exp)).

© www.testbench.in

### **EXAMPLE : Logical**

```

program main ;
  reg a_1,a_0,a_x,a_z;
  reg b_1,b_0,b_x,b_z;
  initial begin
    a_1 = 'b1;a_0 = 'b0;a_x = 'bx;a_z = 'bz;
    b_1 = 'b1;b_0 = 'b0;b_x = 'bx;b_z = 'bz;
    $display("-----");
    $display (" && 0 1 x z ");
    $display("-----");
    $display (" 0  %b  %b  %b  %b ",a_0 && b_0,a_0 && b_1,a_0 && b_x,a_0&& b_z);
    $display (" 1  %b  %b  %b  %b ",a_1 && b_0,a_1 && b_1,a_1 && b_x,a_1&& b_z);
    $display (" x  %b  %b  %b  %b ",a_x && b_0,a_x && b_1,a_x && b_x,a_x&& b_z);
  
```

```

$display (" z   %b  %b  %b ",a_z && b_0,a_z && b_1,a_z && b_x,a_z&& b_z);
$display("-----");
$display("-----");
$display (" || 0  1  x  z ");
$display("-----");
$display (" 0   %b  %b  %b ",a_0 || b_0,a_0 || b_1,a_0 || b_x,a_0 || b_z);
$display (" 1   %b  %b  %b ",a_1 || b_0,a_1 || b_1,a_1 || b_x,a_1 || b_z);
$display (" x   %b  %b  %b ",a_x || b_0,a_x || b_1,a_x || b_x,a_x || b_z);
$display (" z   %b  %b  %b ",a_z || b_0,a_z || b_1,a_z || b_x,a_z || b_z);
$display("-----");
$display("-----");
$display (" ! 0  1  x  z ");
$display("-----");
$display ("   %b  %b  %b  %b ",!b_0,!b_1,!b_x,!b_z);
$display("-----");
end
endprogram

```

## RESULTS

-----  
 $\&\&$  0 1 x z

-----  
0 0 0 0 0  
1 0 1 x x  
x 0 x x x  
z 0 x x x

-----  
|| 0 1 x z

-----  
0 0 1 x x  
1 1 1 1 1  
x x 1 x x  
z x 1 x x

-----  
! 0 1 x z

-----  
1 0 x x

---

## Bitwise :

Operator Token	Operator name
$\sim$	Bitwise negation (unary)
$\&$	Bitwise and (binary)
$\&\sim$	Bitwise nand (binary)
$ $	Bitwise or (binary)
$  \sim$	Bitwise nor (binary)
$\wedge$	Bitwise exclusive or (binary)
$\wedge\sim$ , $\sim\wedge$	Bitwise exclusive nor (binary)

© www.testbench.in

In Systemverilog, bitwise exclusive nor has two notations ( $\wedge\sim$  and  $\sim\wedge$ ).

### EXAMPLE : Bitwise

**program** main ;

```
reg a_1,a_0,a_x,a_z;
reg b_1,b_0,b_x,b_z;
initial begin
    a_1 = 'b1;a_0 = 'b0;a_x = 'bx;a_z = 'bz;
    b_1 = 'b1;b_0 = 'b0;b_x = 'bx;b_z = 'bz;

    $display("-----");
    $display (" ~ 0 1  x  z ");
    $display("-----");
    $display ("   %b  %b  %b  %b ",~b_0,~b_1,~b_x,~b_z);
    $display("-----");
    $display("-----");
    $display (" & 0 1  x  z ");
    $display("-----");
    $display (" 0   %b  %b  %b  %b ",a_0 & b_0,a_0 & b_1,a_0 & b_x,a_0 & b_z);
    $display (" 1   %b  %b  %b  %b ",a_1 & b_0,a_1 & b_1,a_1 & b_x,a_1 & b_z);
```

```

$display (" x  %b  %b  %b  %b ",a_x & b_0,a_x & b_1,a_x & b_x,a_x & b_z);
$display (" z  %b  %b  %b  %b ",a_z & b_0,a_z & b_1,a_z & b_x,a_z & b_z);
$display("-----");
$display("-----");
$display (" &~ 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 &~ b_0,a_0 &~ b_1,a_0 &~ b_x,a_0 &~ b_z);
$display (" 1  %b  %b  %b  %b ",a_1 &~ b_0,a_1 &~ b_1,a_1 &~ b_x,a_1 &~ b_z);
$display (" x  %b  %b  %b  %b ",a_x &~ b_0,a_x &~ b_1,a_x &~ b_x,a_x &~ b_z);
$display (" z  %b  %b  %b  %b ",a_z &~ b_0,a_z &~ b_1,a_z &~ b_x,a_z &~ b_z);
$display("-----");
$display("-----");
$display (" | 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 | b_0,a_0 | b_1,a_0 | b_x,a_0 | b_z);
$display (" 1  %b  %b  %b  %b ",a_1 | b_0,a_1 | b_1,a_1 | b_x,a_1 | b_z);
$display (" x  %b  %b  %b  %b ",a_x | b_0,a_x | b_1,a_x | b_x,a_x | b_z);
$display (" z  %b  %b  %b  %b ",a_z | b_0,a_z | b_1,a_z | b_x,a_z | b_z);
$display("-----");
$display (" |~ 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 |~ b_0,a_0 |~ b_1,a_0 |~ b_x,a_0 |~ b_z);
$display (" 1  %b  %b  %b  %b ",a_1 |~ b_0,a_1 |~ b_1,a_1 |~ b_x,a_1 |~ b_z);
$display (" x  %b  %b  %b  %b ",a_x |~ b_0,a_x |~ b_1,a_x |~ b_x,a_x |~ b_z);
$display (" z  %b  %b  %b  %b ",a_z |~ b_0,a_z |~ b_1,a_z |~ b_x,a_z |~ b_z);
$display("-----");
$display("-----");
$display (" ^ 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 ^ b_0,a_0 ^ b_1,a_0 ^ b_x,a_0 ^ b_z);
$display (" 1  %b  %b  %b  %b ",a_1 ^ b_0,a_1 ^ b_1,a_1 ^ b_x,a_1 ^ b_z);
$display (" x  %b  %b  %b  %b ",a_x ^ b_0,a_x ^ b_1,a_x ^ b_x,a_x ^ b_z);
$display (" z  %b  %b  %b  %b ",a_z ^ b_0,a_z ^ b_1,a_z ^ b_x,a_z ^ b_z);
$display("-----");
$display (" ^~ 0  1  x  z ");
$display("-----");
$display (" 0  %b  %b  %b  %b ",a_0 ^~ b_0,a_0 ^~ b_1,a_0 ^~ b_x,a_0 ^~ b_z);
$display (" 1  %b  %b  %b  %b ",a_1 ^~ b_0,a_1 ^~ b_1,a_1 ^~ b_x,a_1 ^~ b_z);
$display (" x  %b  %b  %b  %b ",a_x ^~ b_0,a_x ^~ b_1,a_x ^~ b_x,a_x ^~ b_z);
$display (" z  %b  %b  %b  %b ",a_z ^~ b_0,a_z ^~ b_1,a_z ^~ b_x,a_z ^~ b_z);

```

```
$display("-----");
end
endprogram
```

## RESULTS

-----  
~ 0 1 x z

-----  
1 0 x x

-----  
& 0 1 x z

-----  
0 0 0 0 0  
1 0 1 x x  
x 0 x x x  
z 0 x x x

-----  
&~ 0 1 x z

-----  
0 0 0 0 0  
1 1 0 x x  
x x 0 x x  
z x 0 x x

-----  
| 0 1 x z

-----  
0 0 1 x x  
1 1 1 1 1  
x x 1 x x  
z x 1 x x

-----  
|~ 0 1 x z

-----  
0 1 0 x x  
1 1 1 1 1  
x 1 x x x  
z 1 x x x

-----  
^ 0 1 x z  
-----

0 0 1 x x  
1 1 0 x x  
x x x x x  
z x x x x  
-----

^~ 0 1 x z  
-----

0 1 0 x x  
1 0 1 x x  
x x x x x  
z x x x x  
-----

### Reduction :

Operator Token	Operator name
&	Unary and
~&	Unary nand
	Unary or
~	Unary nor
^	Unary exclusive or
~^	Unary exclusive

© www.testbench.in

### EXAMPLE : Reduction

```
program main ;
reg [3:0] a_1,a_0,a_01xz,a_1xz,a_0xz,a_Odd1,a_even1;
initial
begin
a_1 = 4'b1111 ;
a_0 = 4'b0000 ;
a_01xz = 4'b01xz ;
a_1xz = 4'b11xz ;
a_0xz = 4'b00xz ;
a_Odd1 = 4'b1110 ;
a_even1 = 4'b1100 ;
```

```

$display("-----");
$display(" a_1 a_0 a_01xz a_1xz a_0xz ");
$display("-----");
$display("& %b %b %b %b
%b ",&a_1,&a_0,&a_01xz,&a_1xz,&a_0xz);
$display(" | %b %b %b %b ",|a_1,|a_0,|a_01xz,|a_1xz,|a_0xz);
$display(" ~& %b %b %b %b
%b ",~&a_1,~&a_0,~&a_01xz,~&a_1xz,~&a_0xz);
$display(" ~| %b %b %b %b ",~|a_1,~|a_0,~|a_01xz,~|a_1xz,~|a_0xz);
$display("-----");
$display("      a_ood1 a_even1 a_1xz");
$display("-----");
$display(" ^ %b %b %b ",^a_0dd1,^a_even1,^a_1xz);
$display(" ~^ %b %b %b ",~^a_0dd1,~^a_even1,~^a_1xz);
$display("-----");
end
endprogram

```

## RESULTS

```

-----
a_1 a_0 a_01xz a_1xz a_0xz
-----
& 1 0 0 x 0
| 1 0 1 1 x
~& 0 1 1 x 1
~| 0 1 0 0 x
-----
a_ood1 a_even1 a_1xz
-----
^ 1 0 x
~^ 0 1 x
-----
```

## Shift :

Operator Token	Operator name
<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right

© www.testbench.in

The left shift operators, << and <<<, shall shift their left operand to the left by the number by the number of bit positions given by the right operand. In both cases, the vacated bit positions shall be filled with zeros. The right shift operators, >> and >>>, shall shift their left operand to the right by the number of bit positions given by the right operand. The logical right shift shall fill the vacated bit positions with zeros. The arithmetic right shift shall fill the vacated bit positions with zeros if the result type is unsigned. It shall fill the vacated bit positions with the value of the most significant (i.e., sign) bit of the left operand if the result type is signed. If the right operand has an x or z value, then the result shall be unknown. The right operand is always treated.

### EXAMPLE :Shift

```
program main ;
  integer a_1,a_0;
  initial begin
    a_1 = 4'b1100 ;
    a_0 = 4'b0011 ;

    $display(" << by 1 a_1 is %b a_0 is %b ",a_1 << 1,a_0 << 1);
    $display(" >> by 2 a_1 is %b a_0 is %b ",a_1 >> 2,a_0 >> 2);
    $display(" <<< by 1 a_1 is %b a_0 is %b ",a_1 <<< 1,a_0 <<< 1);
    $display(" >>> by 2 a_1 is %b a_0 is %b ",a_1 >>> 2,a_0 >>> 2);
  end
endprogram
```

### RESULTS

```
<< by 1 a_1 is 1000 a_0 is 0110
>> by 2 a_1 is 0011 a_0 is 0000
<<< by 1 a_1 is 1000 a_0 is 0110
>>> by 2 a_1 is 1111 a_0 is 0000
```

## **Increment And Decrement :**

```
# ++      increment  
# --      decrement
```

SystemVerilog includes the C increment and decrement assignment operators `++i`, `--i`, `i++`, and `i-`. These do not need parentheses when used in expressions. These increment and decrement assignment operators behave as blocking assignments.

The ordering of assignment operations relative to any other operation within an expression is undefined. An implementation can warn whenever a variable is both written and read-or-written within an integral expression or in other contexts where an implementation cannot guarantee order of evaluation.

For example:

```
i = 10;  
j = i++ + (i = i - 1);
```

After execution, the value of `j` can be 18, 19, or 20 depending upon the relative ordering of the increment and the assignment statements. The increment and decrement operators, when applied to real operands, increment or decrement the operand by 1.0.

### **EXAMPLE : Increment and Decrement**

```
program main ;  
integer a_1,a_0;  
initial begin  
    a_1 = 20;  
    a_0 = 20;  
    a_1 ++;  
    a_0 --;  
    $display (" a_1 is %d a_0 is %d ",a_1,a_0);  
end  
endprogram
```

### **RESULTS**

a\_1 is 21 a\_0 is 19

### **Set :**

```
# inside !inside dist
```

SystemVerilog supports singular value sets and set membership operators.

The syntax for the set membership operator is:

inside\_expression ::= expression inside { open\_range\_list }

The expression on the left-hand side of the inside operator is any singular expression. The set-membership open\_range\_list on the right-hand side of the inside operator is a comma-separated list of expressions or ranges. If an expression in the list is an aggregate array, its elements are traversed by descending into the array until reaching a singular value. The members of the set are scanned until a match is found and the operation returns 1'b1. Values can be repeated, so values and value ranges can overlap. The order of evaluation of the expressions and ranges is non-deterministic.

#### EXAMPLE : Set

```
program main ;
integer i;
initial begin
    i = 20;
    if( i inside {10,20,30})
        $display(" I is in 10 20 30 ");
end
endprogram
```

#### RESULTS

I is in 10 20 30

#### Streaming Operator

The streaming operators perform packing of **bit**-stream types into a **sequence** of bits in a user-specified order.

When used in the left-hand side , the streaming operators perform the **reverse** operation, i.e., unpack a stream of bits into one **or** more variables.

#### Re-Ordering Of The Generic Stream

The stream\_operator << or >> determines the order in which blocks of data are streamed.

>> causes blocks of data to be streamed in left-to-right order

<< causes blocks of data to be streamed in right-to-left order

For Example

```
int j= { "A", "B", "C", "D" };
{ >> {j} } // generates stream "A" "B" "C" "D"
{ << byte {j} } // generates stream "D" "C" "B" "A" (little endian)
{ << 16 {j} } // generates stream "C" "D" "A" "B"
{ << { 8'b0011_0101 } } // generates stream 'b1010_1100 (bit reverse)
{ << 4 { 6'b11_0101 } } // generates stream 'b0101_11
{ >> 4 { 6'b11_0101 } } // generates stream 'b1101_01 (same)
{ << 2 { { << { 4'b1101 } } } } // generates stream 'b1110
```

## Packing Using Streaming Operator

Packing is performed by using the streaming operator on the RHS of the expression.

For example:

```

int j = { "A", "B", "C", "D" };
bit [7:0] arr;
arr = { >> {j} }
arr = { << byte {j} }
arr = { << 16 {j} }
arr = { << { 8'b0011_0101 } }
arr = { << 4 { 6'b11_0101 } }
arr = { >> 4 { 6'b11_0101 } }
arr = { << 2 { { << { 4'b1101 } } } }
```

## Unpacking Using Streaming Operator

UnPacking is performed by using the streaming operator on the LHS of the expression.

For example

```

int a, b, c;
logic [10:0] up [3:0];
logic [11:1] p1, p2, p3, p4;
bit [96:1] y = {>>{ a, b, c }}; // OK: pack a, b, c
int j = {>>{ a, b, c }}; // error: j is 32 bits < 96 bits
bit [99:0] d = {>>{ a, b, c }}; // OK: d is padded with 4 bits
{>>{ a, b, c }} = 23'b1; // error: too few bits in stream
{>>{ a, b, c }} = 96'b1; // OK: unpack a = 0, b = 0, c = 1
{>>{ a, b, c }} = 100'b1; // OK: unpack as above (4 bits unread)
{ >> {p1, p2, p3, p4} } = up; // OK: unpack p1 = up[3], p2 = up[2],
// p3 = up[1], p4 = up[0]
```

## Streaming Dynamically Sized Data

```

Stream = {<< byte{p.header, p.len, p.payload, p.crc}}; // packing
Stream = {<<byte{p.header, p.len, p.payload with [0 +: p.len], p.crc}};
{<< byte{ p.header, p.len, p.payload with [0 +: p.len], p.crc }} = stream; //unpacking
q = {<<byte{p}}; // packing all the contents of an object.( p is a object )
```

## OPERATOR PRECEDENCY

( ) Highest precedence

++ --

& ~& | ~| ^ ~^ ~ >< -

(unary)

\* / %

+ -

<< >>

< <= > >= in !in dist

```
=?= != == != === !=  
& &~  
^ ^~  
| |~  
&&  
||  
?:  
= += -= *= /= %=  
<<= >>= &= |= ^= ~&= ~|= ~^= Lowest precedence
```

## **EVENTS**

An identifier declared as an event data type is called a named event. Named event is a data type which has no storage. In verilog, a named event can be triggered explicitly using "->" . Verilog Named Event triggering occurrence can be recognized by using the event control "@" . Named events and event control give a powerful and efficient means of describing the communication between, and synchronization of, two or more concurrently active processes.

SystemVerilog named events support the same basic operations as verilog named event, but enhance it in several ways.

### **Triggered**

The "triggered" event property evaluates to true if the given event has been triggered in the current time-step and false otherwise. If event\_identifier is null, then the triggered event property evaluates to false. Using this mechanism, an event trigger shall unblock the waiting process whether the wait executes before or at the same simulation time as the trigger operation.

In the following example, event "e" is triggered at time 20,40,60,80 . So the Value of "e.triggered" should be TRUE at time 20,40,60,80 and FALSE at rest of the time.

### **EXAMPLE:**

```
module main;  
event e;  
initial  
repeat(4)  
begin  
#20;  
->e ;  
$display(" e is triggered at %t ",$time);  
end  
initial  
#100 $finish;  
always  
begin  
#10;  
if(e.triggered)
```

```

$display(" e is TRUE at %t",$time);
else
    $display(" e is FALSE at %t",$time);
end
endmodule

```

## RESULT

e is FALSE at	10
e is triggered at	20
e is TRUE at	20
e is FALSE at	30
e is triggered at	40
e is TRUE at	40
e is FALSE at	50
e is triggered at	60
e is TRUE at	60
e is FALSE at	70
e is triggered at	80
e is TRUE at	80
e is FALSE at	90

## Wait()

In SystemVerilog , Named Event triggering occurrence can also be recognized by using the event control wait(). Wait() statement gets blocked until it evaluates to TRUE. As we have seen in the previous example, that "event\_name.triggered" returns the triggering status of the event in the current time step.

### EXAMPLE:

```

module event_m;
    event a;
    initial
        repeat(4)
            #20 -> a;
        always
            begin
                @a;
                $display(" ONE :: EVENT A is triggered ");
            end
        always
            begin
                wait(a.triggered);
                $display(" TWO :: EVENT A is triggered ");
            end

```

```
    #1;  
end  
endmodule
```

## RESULT:

```
ONE :: EVENT A is triggered  
TWO :: EVENT A is triggered  
ONE :: EVENT A is triggered  
TWO :: EVENT A is triggered  
ONE :: EVENT A is triggered  
TWO :: EVENT A is triggered  
ONE :: EVENT A is triggered  
TWO :: EVENT A is triggered
```

## Race Condition

For a trigger to unblock a process waiting on an event, the waiting process must execute the @ statement before the triggering process executes the trigger operator, ->. If the trigger executes first, then the waiting process remains blocked.

Using event\_name.triggered statement, an event trigger shall unblock the waiting process whether the wait executes before or at the same simulation time as the trigger operation. The triggered event property, thus, helps eliminate a common race condition that occurs when both the trigger and the wait (using @) happen at the same time. A process that blocks waiting for an event might or might not unblock, depending on the execution order of the waiting and triggering processes (race condition) . However, a process that waits on the triggered state always unblocks, regardless of the order of execution of the wait and trigger operations.

In the following example, event "e1" is triggered and a process is waiting on "e1" in the same time step. The process can never catch the triggering of "e1" as it occurs after the event "e1" triggering. Event "e2" triggering occurrence can be recognized by wait (e2.triggered) in spite of the above condition.

## EXAMPLE:

```
module main;  
  event e1,e2;  
  initial  
    repeat(4)  
    begin  
      #20;  
      ->e1 ;
```

```

@(e1)
$display(" e1 is triggered at %t ",$time);
end
initial
repeat(4)
begin
#20;
->e2 ;
wait(e2.triggered);
$display(" e2 is triggered at %t ",$time);
end
endmodule

```

## RESULT

e2 is triggered at	20
e2 is triggered at	40
e2 is triggered at	60
e2 is triggered at	80

## Nonblocking Event Trigger

Nonblocking events are triggered using the ->> operator. The effect of the ->> operator is that the statement executes without blocking and it creates a nonblocking assign update event in the time in which the delay control expires, or the event-control occurs. The effect of this update event shall be to trigger the referenced event in the nonblocking assignment region of the simulation cycle.

## Merging Events

An event variable can be assigned to another event variable. When a event variable is assigned to other , both the events point to same synchronization object. In the following example, Event "a" is assigned to event "b" and when event "a" is triggered, event occurrence can be seen on event "b" also.

## EXAMPLE:

```

module events_ab;
event a,b;
initial begin
#1 -> b; // trigger both always blocks
-> a;
#10 b = a; // merge events
#20 -> a; // both will trigger , 3 trigger events but have 4 trigger responses.
end
always@(a) begin
$display(" EVENT A is triggered ");
#20;

```

```

end
always@(b) begin
    $display(" EVENT B is triggered ");
    #20;
end
endmodule

```

### **RESULTS:**

EVENT B is triggered  
 EVENT A is triggered  
 EVENT B is triggered  
 EVENT A is triggered

When events are merged, the assignment only affects the execution of subsequent event control or wait operations. If a process is blocked waiting for event1 when another event is assigned to event1, the currently waiting process shall never unblock. In the following example, "always@(b)" is waiting for the event on "b" before the assignment "b = a" and this waiting always block was never unblocked.

### **EXAMPLE:**

```

module events_ab;
    event a,b;
    initial
    begin
        #20 -> a;
        b = a;
        #20 -> a;
    end
    always@(a)
        $display(" EVENT A is triggered ");
    always@(b)
        $display(" EVENT B is also triggered ");
    endmodule

```

### **RESULTS:**

EVENT A is triggered  
 EVENT A is triggered

### **Null Events**

SystemVerilog event variables can also be assigned a null object, when assigned null to event variable, the association between the synchronization object and the event variable is broken.

### **EXAMPLE:**

```

program main;
    event e;
    initial

```

```

begin
  repeat(4)
    #($random()%10) -> e;
    e = null;
  repeat(4)
    #($random()%10) -> e;
end
initial
  forever
    begin
      @e ;
      $display(" e is triggered at %t",$time);
    end
endprogram

```

#### **RESULT:**

```

e is triggered at      348
e is triggered at      4967
e is triggered at     9934
e is triggered at    14901
** ERROR ** Accessed Null object

```

#### Wait Sequence

The wait\_order construct suspends the calling process until all of the specified events are triggered in the given order (left to right) or any of the un-triggered events are triggered out of order and thus causes the operation to fail. Wait\_order() does not consider time, only ordering is considered.

#### **EXAMPLE:**

```

module main;
  event e1,e2,e3;
  initial
  begin
    #10;
    -> e1;
    -> e2;
    -> e3;
    #10;
    -> e3;
    -> e1;
    -> e2;
    #10;
    -> e3;

```

```

-> e2;
-> e3;
end
always
begin
  wait_order(e1,e2,e3)
    $display(" Events are in order ");
  else
    $display(" Events are out of order ");
end
endmodule

```

### **RESULT:**

Events are in order  
 Events are out of order  
 Events are out of order

### **Events Comparison**

Event variables can be compared against other event variables or the special value null. Only the following operators are allowed for comparing event variables:

- Equality (==) with another event or with null.
- Inequality (!=) with another event or with null.
- Case equality (====) with another event or with null (same semantics as ==).
- Case inequality (!==) with another event or with null (same semantics as !=).
- Test for a Boolean value that shall be 0 if the event is null and 1 otherwise.

### **EXAMPLE:**

```

module main;
  event e1,e2,e3,e4;
  initial
  begin
    e1 = null;
    e2 = e3;
    if(e1)
      $display(" e1 is not null ");
    else
      $display(" e1 is null ");
    if(e2)
      $display(" e2 is not null");
    else
      $display(" e2 is null");
    if(e3 == e4)
      $display( " e3 and e4 are same events ");
  
```

```

else
    $display( " e3 and e4 are not same events ");
if(e3 == e2)
    $display( " e3 and e2 are same events ");
else
    $display( " e3 and e2 are not same events ");
end
endmodule

```

#### **RESULT:**

e1 is null  
e2 is not null  
e3 and e4 are not same events  
e3 and e2 are same events

### **CONTROL STATEMENTS**

#### **Sequential Control:**

Statements inside sequential control constructs are executed sequentially.

- if-else Statement
- case Statement
- repeat loop
- for loop
- while loop
- do-while
- foreach
- Loop Control
- randcase Statements

**if-else Statement :** The if-else statement is the general form of selection statement.

**case Statement :** The case statement provides for multi-way branching.

**repeat loop :** Repeat statements can be used to repeat the execution of a statement or statement block a fixed number of times.

**for loop :** The for construct can be used to create loops.

**while loop :** The loop iterates while the condition is true.

**do-while :** condition is checked after loop iteration.

**foreach :** foreach construct specifies iteration over the elements of a single dimensional

fixed-size arrays, dynamic arrays and SmartQs.

Loop Control : The break and continue statements are used for flow control within loops.

**EXAMPLE : if**

```
program main ;
    integer i;
    initial begin
        i = 20;
        if( i == 20)
            $display(" I is equal to %d ",i);
        else
            $display(" I is not equal to %d ",i);
    end
endprogram
```

**RESULTS**

I is equal to 20 **EXAMPLE : case and repeat**

```
program main ;
    integer i;
    initial begin
        repeat(10)begin
            i = $random();
            case(1) begin
                (i<0) :$display(" i is less than zero i==%d\n",i);
                (i>0) :$display(" i is grater than zero i=%d\n",i);
                (i == 0):$display(" i is equal to zero i=%d\n",i);
            end
        end
    end
endprogram
```

**RESULTS**

```
i is grater than zero i=69120
i is grater than zero i=475628600
i is grater than zero i=1129920902
i is grater than zero i=773000284
i is grater than zero i=1730349006
i is grater than zero i=1674352583
i is grater than zero i=1662201030
i is grater than zero i=2044158707
i is grater than zero i=1641506755
i is grater than zero i=797919327
```

### **EXAMPLE : forloop**

```
program for_loop;
integer count, i;
initial begin
  for(count = 0, i=0; i*count<50; i++, count++)
    $display("Value i = %0d\n", i);
end
endprogram
```

### **RESULTS**

```
Value i = 0
Value i = 1
Value i = 2
Value i = 3
Value i = 4
Value i = 5
Value i = 6
Value i = 7
```

### **EXAMPLE : whileloop**

```
program while_loop;
integer operator=0;
initial begin
  while (operator<5)begin
    operator += 1;
    $display("Operator is %0d\n", operator);
  end
end
endprogram
```

### **RESULTS**

```
Operator is 1
Operator is 2
Operator is 3
Operator is 4
Operator is 5
```

### **EXAMPLE : dowhile**

```
program test;
integer i=0;
initial begin
  do
    begin
      $display("i = %0d \n", i);
    end
  end
end
```

```

    i++;
end
while (i < 10);
end
endprogram

```

## RESULTS

```

i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9

```

The foreach construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array (fixed-size, dynamic, or associative) followed by a list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array. The foreach construct is similar to a repeat loop that uses the array bounds to specify the repeat count instead of an expression.

The mapping of loop variables to array indexes is determined by the dimension cardinality, as described in multidimensional topic.

The foreach arranges for higher cardinality indexes to change more rapidly.

```

// 1 2 3      3 4   1 2 -> Dimension numbers
int A [2][3][4]; bit [3:0][2:1] B [5:1][4];
foreach( A [ i, j, k ] ) ...
foreach( B [ q, r, , s ] ) ...

```

The first foreach causes i to iterate from 0 to 1, j from 0 to 2, and k from 0 to 3. The second foreach causes q to iterate from 5 to 1, r from 0 to 3, and s from 2 to 1 (iteration over the third index is skipped).

## EXAMPLE : foreach

```

program example;
  string names[$]={ "Hello", "SV"};
  int fxd_arr[2][3] = '{'{'1,2,3'},'{4,5,6'}};
  initial begin
    foreach (names[i])
      $display("Value at index %0d is %0s\n", i, names[i]);
    foreach(fxd_arr[.j])
      $display(fxd_arr[1][j]);end

```

**endprogram**

## RESULTS

Value at index 0 is Hello

Value at index 1 is SV

4  
5  
6

## EXAMPLE : randcase

```
program rand_case;
    integer i;
initial begin
repeat(10)begin
    randcase
    begin
        10: i=1;
        20: i=2;
        50: i=3;
    end
    $display(" i is %d \n",i);end
end
endprogram
```

## RESULTS

i is 3  
i is 2  
i is 3  
i is 3  
i is 3  
i is 3  
i is 1  
i is 1  
i is 1  
i is 2

## Enhanced For Loop

In Verilog, the variable used to control a for loop must be declared prior to the loop. If loops in two or more parallel procedures use the same loop control variable, there is a potential of one loop modifying the variable while other loops are still using it.

SystemVerilog adds the ability to declare the for loop control variable within the for loop. This creates a local variable within the loop. Other parallel loops cannot inadvertently affect the loop control variable.

For example:

```
module foo;
    initial begin
        for (int i = 0; i <= 255; i++)
            ...
    end
    initial begin
        loop2: for (int i = 15; i >= 0; i--)
            ...
    end
endmodule
```

### Unique:

A unique if asserts that there is no overlap in a series of if...else...if conditions, i.e., they are mutually exclusive and hence it is safe for the expressions to be evaluated in parallel. In a unique if, it shall be legal for a condition to be evaluated at any time after entrance into the series and before the value of the condition is needed. A unique if shall be illegal if, for any such interleaving of evaluation and use of the conditions, more than one condition is true. For an illegal unique if, an implementation shall be required to issue a warning, unless it can demonstrate a legal interleaving so that no more than one condition is true.

### EXAMPLE :

```
module uniq;
    initial
    begin
        for (int a = 0;a< 6;a++)
            unique if ((a==0) || (a==1) ) $display("0 or 1");
            else if (a == 2) $display("2");
            else if (a == 4) $display("4"); // values 3,5,6 cause a warning
    end
endmodule
```

### RESULTS:

0 or 1

0 or 1

2

RT Warning: No condition matches in 'unique if' statement.

4

RT Warning: No condition matches in 'unique if' statement.

### **Priority:**

A priority if indicates that a series of if...else...if conditions shall be evaluated in the order listed. In the preceding example, if the variable a had a value of 0, it would satisfy both the first and second conditions, requiring priority logic. An implementation shall also issue a warning if it determines that no condition is true, or it is possible that no condition is true, and the final if does not have a corresponding else.

#### **EXAMPLE:**

```
module prioriti;
initial
  for(int a = 0;a<7;a++)
    priority if (a[2:1]==0) $display("0 or 1");
    else if (a[2] == 0) $display("2 or 3");
    else $display("4 to 7"); //covers all other possible values, so no warning
endmodule
```

#### **RESULTS:**

```
0 or 1
0 or 1
2 or 3
2 or 3
4 to 7
4 to 7
4 to 7
```

If the case is qualified as priority or unique, the simulator shall issue a warning message if no case item matches. These warnings can be issued at either compile time or run time, as soon as it is possible to determine the illegal condition.

#### **EXAMPLE:**

```
module casee;
initial
  begin
    for(int a = 0;a<4;a++)
      unique case(a) // values 3,5,6,7 cause a warning
        0,1: $display("0 or 1");
        2: $display("2");
        4: $display("4");
  endcase
  for(int a = 0;a<4;a++)
    priority casez(a) // values 4,5,6,7 cause a warning
      3'b00?: $display("0 or 1");
```

```
3'b0??: $display("2 or 3");
endcase
end
endmodule
```

## RESULTS:

0 or 1

0 or 1

2

Warning: No condition matches in 'unique case' statement.

0 or 1

0 or 1

2 or 3

2 or 3

## **PROGRAM BLOCK**

The module is the basic building block in Verilog which works well for Design. However, for the testbench, a lot of effort is spent getting the environment properly initialized and synchronized, avoiding races between the design and the testbench, automating the generation of input stimuli, and reusing existing models and other infrastructure.

Systemverilog adds a new type of block called program block. It is declared using program and endprogram keywords.

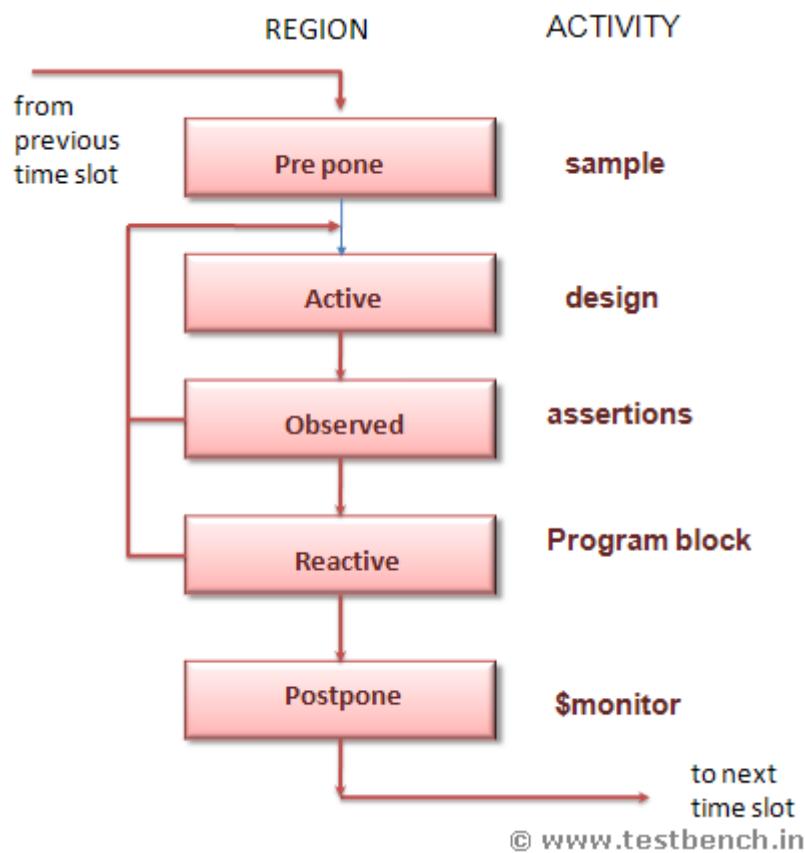
The program block serves these basic purposes:

- > Separates the testbench from the DUT.
- > The program block helps ensure that test bench transitions do not have race conditions with the design
- > It provides an entry point to the execution of testbenches.
- > It creates a scope that encapsulates program-wide data.
- > It provides a syntactic context that specifies scheduling in the Reactive region which avoids races.
- > It does not allow always block. Only initial and methods are allowed, which are more controllable.
- > Each program can be explicitly exited by calling the \$exit system task. Unlike \$finish, which exits simulation immediately, even if there are pending events.
- > Just like a module, program block has ports. One or more program blocks can be instantiated in a top-level netlist, and connected to the DUT.

The program construct serves as a clear separator between design and testbench, and, more importantly, it specifies specialized execution semantics in the Reactive region for all elements declared within the program. Together with clocking blocks, the program construct provides for race-free interaction between the design and the testbench, and enables cycle and transaction level abstractions.

For example:

```
program test (input clk, input [16:1] addr, inout [7:0] data);
    initial ...
endprogram
program test ( interface device_ifc );
    initial ...
endprogram
```



program schedules events in the Reactive region, the clocking block construct is very useful to automatically sample the steady-state values of previous time steps or clock cycles. Programs that read design values exclusively through clocking blocks with #0 input skews are insensitive to read-write races. It is important to note that simply sampling input signals (or setting non-zero skews on clocking block inputs) does not eliminate the potential for races. Proper input sampling only addresses a single clocking block. With multiple clocks, the arbitrary order in which overlapping or simultaneous clocks are processed is still a potential source for races.

Following example demonstrates the difference between the module based testbench and program based testbenches

```

module DUT();
  reg q = 0;
  reg clk = 0;
  initial
    #10 clk = 1;
  always @(posedge clk)
    q <= 1;
  endmodule

module Module_based_TB();
  always @(posedge DUT.clk) $display("Module_based_TB : q = %b\n", DUT.q);
  endmodule

program Program_based_TB();
  initial
    forever @(posedge DUT.clk) $display("Program_based_TB : q = %b\n", DUT.q);
  endprogram

```

#### **RESULT:**

Module\_based\_TB : q = 0  
 program\_based\_TB : q = 1

### **PROCEDURAL BLOCKS**

#### **Final:**

Verilog procedural statements are in initial or always blocks, tasks, or functions. SystemVerilog adds a final block that executes at the end of simulation. SystemVerilog final blocks execute in an arbitrary but deterministic sequential order. This is possible because final blocks are limited to the legal set of statements allowed for functions.

#### **EXAMPLE :**

```

module fini;
  initial
    #100 $finish;
  final
    $display(" END OF SIMULATION at %d ",$time);
  endmodule

```

#### **RESULTS:**

END OF SIMULATION at 100

#### **Jump Statements:**

SystemVerilog has statements to control the loop statements.

break : to go out of loop as C

continue : skip to end of loop as C

return expression : exit from a function

return : exit from a task or void function

## Event Control:

Any change in a variable or net can be detected using the @ event control, as in Verilog. If the expression evaluates to a result of more than 1 bit, a change on any of the bits of the result (including an x to z change) shall trigger the event control.

SystemVerilog adds an iff qualifier to the @ event control.

### EXAMPLE:

```
module latch (output logic [31:0] y, input [31:0] a, input enable);
    always @(a iff enable == 1)
        y <= a; //latch is in transparent mode
    endmodule
```

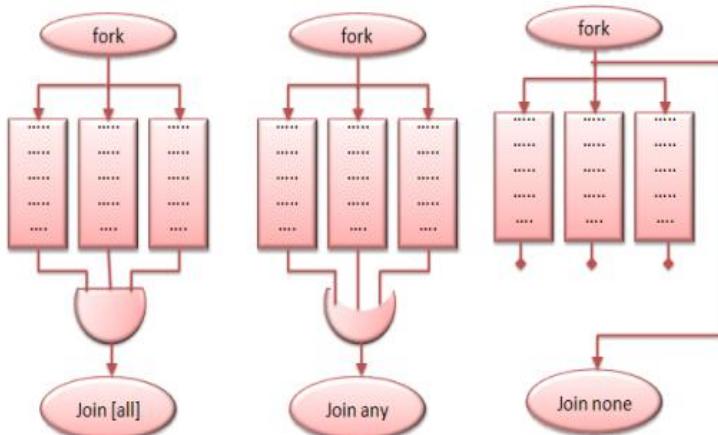
### Always:

In an always block that is used to model combinational logic, forgetting an else leads to an unintended latch. To avoid this mistake, SystemVerilog adds specialized always\_comb and always\_latch blocks, which indicate design intent to simulation, synthesis, and formal verification tools. SystemVerilog also adds an always\_ff block to indicate sequential logic.

### EXAMPLE:

```
always_comb
    a = b & c;
always_latch
    if(ck) q <= d;
always_ff @(posedge clock iff reset == 0 or posedge reset)
    r1 <= reset ? 0 : r2 + 1;FORK JOIN
```

A Verilog fork...join block always causes the process executing the fork statement to block until the termination of all forked processes. With the addition of the join\_any and join\_none keywords, SystemVerilog provides three choices for specifying when the parent (forking)



process resumes execution.

## Fork Join None

The parent process continues to execute concurrently with all the processes spawned by the fork. The spawned processes do not start executing until the parent thread executes a blocking statement.

EXAMPLE : fork/join none

```
program main ;
initial
begin
#10;
$display(" BEFORE fork time = %d ",$time );
fork
begin
#(20);
$display("time = %d # 20 ",$time );
end
begin
#(10);
$display("time = %d # 10 ",$time );
end
begin
#(5);
$display("time = %d # 5 ",$time );
end
join_none
$display(" time = %d Outside the main fork ",$time );
#(40);
end
endprogram
```

## RESULTS

```
BEFORE fork time =          10
time =      10 Outside the main fork
time =      15 # 5
time =      20 # 10
time =      30 # 20
```

## Fork Join Any

The parent process blocks until any one of the processes spawned by this fork completes.

EXAMPLE : fork/join any

```
program main;
initial begin
#(10);
```

```

$display(" BEFORE fork time = %d ",$time );
fork
  begin
    #(20);
    $display("time = %d # 20 ",$time );
  end
  begin
    #(10);
    $display("time = %d # 10 ",$time );
  end
  begin
    #(5);
    $display("time = %d # 5 ",$time );
  end
join_any
$display(" time = %d Outside the main fork ",$time );
#(40);
end
endprogram

```

## RESULTS

```

BEFORE fork time =          10
time =          15 # 5
time =          15 Outside the main fork
time =          20 # 10
time =          30 # 20

```

### For Join All

The parent process blocks until all the processes spawned by this fork complete.

#### EXAMPLE : fork/join all

```

program main ;
initial
begin
  #(10);
  $display(" BEFORE fork time = %d ",$time );
fork
  begin
    #(20);
    $display("time = %d # 20 ",$time );
  end
  begin
    #(10);

```

```

$display("time = %d # 10 ",$time );
end
begin
  #(5);
  $display("time = %d # 5 ",$time );
end
join
$display(" time = %d Outside the main fork ",$time );
#(40);
end
endprogram

```

## RESULTS

```

BEFORE fork  time =          10
time =      15 # 5
time =      20 # 10
time =      30 # 20
time =      30 Outside the main fork

```

When defining a fork/join block, encapsulating the entire fork inside begin..end, results in the entire block being treated as a single thread, and the code executes consecutively.

### EXAMPLE : sequential statement in fork/join

```

program main ;
initial begin
  #(10);
  $display(" First fork  time = %d ",$time );
  fork
    begin
      # (20);
      $display("time = %d # 20 ",$time);
    end
    begin
      #(10);
      $display("time = %d # 10 ",$time);
    end
    begin
      #(5);
      $display("time = %d # 5 ",$time);
      #(2);
      $display("time = %d # 2 ",$time);
    end
  join_any

```

```

$display(" time = %d Outside the main fork ",$time );
#(40);
end
endprogram

```

#### **RESULTS:**

```

First fork time =          10
time =      15 # 5
time =      17 # 2
time =      17 Outside the main fork
time =      20 # 10
time =      30 # 20

```

### **FORK CONTROL**

#### **Wait Fork Statement**

The wait fork statement is used to ensure that all immediate child subprocesses (processes created by the current process, excluding their descendants) have completed their execution.

#### **EXAMPLE**

```

program main();
initial begin
  #(10);
  $display(" BEFORE fork time = %0d ",$time );
  fork
    begin
      # (20);
      $display(" time = %0d # 20 ",$time );
    end
    begin
      #(10);
      $display(" time = %0d # 10 ",$time );
    end
    begin
      #(5);
      $display(" time = %0d # 5 ",$time );
    end
  join_any
  $display(" time = %0d Outside the main fork ",$time );
end
endprogram

```

#### **RESULTS**

BEFORE fork time = 10

```
time = 15 # 5  
time = 15 Outside the main fork
```

In the above example, Simulation ends before the #10 and #20 gets executed. In some situations, we need to wait until all the threads got finished to start the next task. Using wait fork, will block the till all the child processes complete.

**EXAMPLE:**

```
program main();  
initial begin  
    #(10);  
    $display(" BEFORE fork time = %0d ",$time );  
fork  
    begin  
        # (20);  
        $display(" time = %0d # 20 ",$time );  
    end  
    begin  
        #(10);  
        $display(" time = %0d # 10 ",$time );  
    end  
    begin  
        #(5);  
        $display(" time = %0d # 5 ",$time );  
    end  
    join_any  
    $display(" time = %0d Outside the main fork ",$time );  
    wait fork ;  
    $display(" time = %0d After wait fork ",$time );  
  
end  
endprogram
```

**RESULTS**

```
BEFORE fork time = 10  
time = 15 # 5  
time = 15 Outside the main fork  
time = 20 # 10  
time = 30 # 20  
time = 30 After wait fork
```

## **Disable Fork Statement**

The disable fork statement terminates all active descendants (subprocesses) of the calling process.

In other words, if any of the child processes have descendants of their own, the disable fork statement shall terminate them as well. Sometimes, it is required to kill the child processes after certain condition.

### **EXAMPLE**

```
program main();
initial begin
  #(10);
  $display(" BEFORE fork time = %0d ",$time );
  fork
    begin
      # (20);
      $display(" time = %0d # 20 ",$time );
    end
    begin
      #(10);
      $display(" time = %0d # 10 ",$time );
    end
    begin
      #(5);
      $display(" time = %0d # 5 ",$time );
    end
  join_any
  $display(" time = %0d Outside the main fork ",$time );
end
```

```
initial
#100 $finish;
endprogram
```

### **RESULTS**

```
BEFORE fork time = 10
time = 15 # 5
time = 15 Outside the main fork
time = 20 # 10
time = 30 # 20
```

In the following example, disable for kills the threads #10 and #20.

## EXAMPLE

```
program main();
initial begin
  #(10);
  $display(" BEFORE fork time = %0d ",$time );
fork
begin
  # (20);
  $display(" time = %0d # 20 ",$time );
end
begin
  #(10);
  $display(" time = %0d # 10 ",$time );
end
begin
  #(5);
  $display(" time = %0d # 5 ",$time );
end
join_any
$display(" time = %0d Outside the main fork ",$time );
disable fork;
$display(" Killed the child processes");
end
```

## initial

```
#100 $finish;
endprogram
```

## RESULTS

```
BEFORE fork time = 10
time = 15 # 5
time = 15 Outside the main fork
Killed the child processes
```

## SUBROUTINES

### Begin End

With SystemVerilog, multiple statements can be written between the task declaration and endtask, which means that the begin .... end can be omitted. If begin .... end is omitted, statements are executed sequentially, the same as if they were enclosed in a begin .... end group. It shall also be legal to have no statements at all.

## Tasks:

A Verilog task declaration has the formal arguments either in parentheses or in declaration.

```
task mytask1 (output int x, input logic y);
```

With SystemVerilog, there is a default direction of input if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments a and b default to inputs, and u and v are both outputs.

```
task mytask3(a, b, output logic [15:0] u, v);
```

## Return In Tasks

In Verilog, a task exits when the endtask is reached. With SystemVerilog, the return statement can be used to exit the task before the endtask keyword.

In the following example, Message "Inside Task : After return statement" is not executed because the task exited before return statement.

### **EXAMPLE**

```
program main();
  task task_return();
    $display("Inside Task : Before return statement");
    return;
    $display("Inside Task : After return statement");
  endtask
initial
  task_return();
endprogram
```

### **RESULT:**

Inside Task : Before return statement

## Functions:

```
function logic [15:0] myfunc1(int x, int y);
```

Function declarations default to the formal direction input if no direction has been specified.

Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments a and b default to inputs, and u and v are both outputs:

```
function logic [15:0] myfunc3(int a, int b, output logic [15:0] u, v);
```

## Return Values And Void Functions::

SystemVerilog allows functions to be declared as type void, which do not have a return value. For nonvoid functions, a value can be returned by assigning the function name to a value, as in Verilog, or by using return with a value. The return statement shall override any value assigned to the function name. When the return statement is used, nonvoid functions must specify an expression with the return.

### **EXAMPLE:**

```
function [15:0] myfunc2 (input [7:0] x,y);
  return x * y - 1; //return value is specified using return statement
endfunction
```

```
// void functions  
function void myprint (int a);
```

**Pass By Reference:** In verilog, method arguments takes as pass by value. The inputs are copied when the method is called and the outputs are assigned to outputs when exiting the method. In SystemVerilog , methods can have pass by reference. Arguments passed by reference are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data via the reference.

In the following example, variable a is changed at time 10,20 and 30. The method pass\_by\_val , copies only the value of the variable a, so the changes in variable a which are occurred after the task pass\_by\_val call, are not visible to pass\_by\_val. Method pass\_by\_ref is directly referring to the variable a. So the changes in variable a are visible inside pass\_by\_ref.

**EXAMPLE:**

```
program main();  
int a;  
initial  
  begin  
    #10 a = 10;  
    #10 a = 20;  
    #10 a = 30;  
    #10 $finish;  
  end  
task pass_by_val(int i);  
  forever  
    @i $display("pass_by_val: I is %0d",i);  
  endtask  
task pass_by_ref(ref int i);  
  forever  
    @i $display("pass_by_ref: I is %0d",i);  
  endtask  
initial  
  pass_by_val(a);  
initial  
  pass_by_ref(a);  
endprogram
```

**RESULT**

```
pass_by_ref: I is 10  
pass_by_ref: I is 20  
pass_by_ref: I is 30
```

### **Default Values To Arguments:**

SystemVerilog allows to declare default values to arguments. When the subroutines are called, arguments those are omitted, will take default value.

#### **EXAMPLE:**

```
program main();
    task display(int a = 0,int b,int c = 1 );
        $display("%0d %0d %0d ",a,b,c);
    endtask
    initial
    begin
        display( , 5 ); // is equivalent to display( 0, 5, 1 );
        display( 2, 5 ); // is equivalent to display( 2, 5, 1 );
        display( , 5, ); // is equivalent to display( 0, 5, 1 );
        display( , 5, 7 ); // is equivalent to display( 0, 5, 7 );
        display( 1, 5, 2 ); // is equivalent to display( 1, 5, 2 );
    end
endprogram
```

#### **RESULT:**

```
0 5 1
2 5 1
0 5 1
0 5 7
1 5 2
```

### **Argument Binding By Name**

SystemVerilog allows arguments to tasks and functions to be bound by name as well as by position. This allows specifying non-consecutive default arguments and easily specifying the argument to be passed at the call.

#### **EXAMPLE:**

```
program main();
    function void fun( int j = 1, string s = "no" );
        $display("j is %0d : s is %s ",j,s);
    endfunction
    initial
    begin
        fun( .j(2), .s("yes") ); // fun( 2, "yes" );
        fun( .s("yes") ); // fun( 1, "yes" );
        fun( , "yes" ); // fun( 1, "yes" );
        fun( .j(2) ); // fun( 2, "no" );
        fun( .s("yes"), .j(2) ); // fun( 2, "yes" );
    end
endprogram
```

```
fun( .s(), .j() ); // fun( 1, "no" );
fun( 2 ); // fun( 2, "no" );
fun(); // fun( 1, "no" );
end
```

**endprogram**

**RESULT**

```
j is 2 : s is yes
j is 1 : s is yes
j is 1 : s is yes
j is 2 : s is no
j is 2 : s is yes
j is 1 : s is no
j is 2 : s is no
j is 1 : s is no
```

### Optional Argument List

When a task or function specifies no arguments, the empty parenthesis, (), following the task/function name shall be optional. This is also true for tasks or functions that require arguments, when all arguments have defaults specified.

**EXAMPLE**

```
program main();
  function void fun();
    $display("Inside function");
  endfunction
  initial
  begin
    fun();
    fun;
  end
endprogram
```

**RESULT**

```
Inside function
Inside function
```

### SEMAPHORE

Conceptually, a semaphore is a bucket. When a semaphore is allocated, a bucket that contains a fixed number of keys is created. Processes using semaphores must first procure a key from the bucket before they can continue to execute. If a specific process requires a key, only a fixed number of occurrences of that process can be in progress simultaneously. All others must wait until a sufficient number of keys is returned to the bucket. Semaphores are typically used for mutual exclusion, access control to shared resources, and basic synchronization.

Semaphore is a built-in class that provides the following methods:

- Create a semaphore with a specified number of keys: new()
- Obtain one or more keys from the bucket: get()
- Return one or more keys into the bucket: put()
- Try to obtain one or more keys without blocking: try\_get()

#### EXAMPLE:semaphore

```
program main ;
semaphore sema = new(1);
initial begin
repeat(3) begin
    fork
        ////////////// PROCESS 1 /////////////
        begin
            $display("1: Waiting for key");
            sema.get(1);
            $display("1: Got the Key");
            #(10);// Do some work
            sema.put(1);
            $display("1: Returning back key ");
            #(10);
        end
        ////////////// PROCESS 2 /////////////
        begin
            $display("2: Waiting for Key");
            sema.get(1);
            $display("2: Got the Key");
            #(10);//Do some work
            sema.put(1);
            $display("2: Returning back key ");
            #(10);
        end
    join
end
#1000;
end
endprogram
```

## **RESULTS:**

1: Waiting for key  
1: Got the Key  
2: Waiting for Key  
1: Returning back key  
2: Got the Key  
2: Returning back key  
1: Waiting for key  
1: Got the Key  
2: Waiting for Key  
1: Returning back key  
2: Got the Key  
2: Returning back key  
1: Waiting for key  
1: Got the Key  
2: Waiting for Key  
1: Returning back key  
2: Got the Key  
2: Returning back key  
1: Waiting for key  
1: Got the Key  
2: Waiting for Key  
1: Returning back key  
2: Got the Key  
2: Returning back key

## **MAILBOX**

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.

Mailbox is a built-in class that provides the following methods:

- Create a mailbox: new()
- Place a message in a mailbox: put()
- Try to place a message in a mailbox without blocking: try\_put()
- Retrieve a message from a mailbox: get() or peek()
- Try to retrieve a message from a mailbox without blocking: try\_get() or try\_peek()
- Retrieve the number of messages in the mailbox: num()

## **EXAMPLE:**

```
program meanin ;  
mailbox my_mailbox;  
initial begin  
    my_mailbox = new();  
    if (my_mailbox)  
        begin  
            fork  
                put_packets();  
                get_packets();  
                #10000;
```

```

join_any
end
 #(1000);
$display("END of Program");
end
task put_packets();
integer i;
begin
  for (i=0; i<10; i++)
begin
  #(10);
  my_mailbox.put(i);
  $display("Done putting packet %d @time %d",i, $time);
end
end
endtask
task get_packets();
integer i,packet;
begin
  for (int i=0; i<10; i++)
begin
  my_mailbox.get(packet);
  $display("Got packet %d @time %d", packet, $time);
end
end
endtask
endprogram

```

#### RESULTS:

Done putting packet	0 @time	10
Got packet	0 @time	10
Done putting packet	1 @time	20
Got packet	1 @time	20
Done putting packet	2 @time	30
Got packet	2 @time	30
Done putting packet	3 @time	40
Got packet	3 @time	40
Done putting packet	4 @time	50
Got packet	4 @time	50
Done putting packet	5 @time	60
Got packet	5 @time	60

Done putting packet	6 @time	70
Got packet	6 @time	70
Done putting packet	7 @time	80
Got packet	7 @time	80
Done putting packet	8 @time	90
Got packet	8 @time	90
Done putting packet	9 @time	100
Got packet	9 @time	100
END of Program		

## **FINE GRAIN PROCESS CONTROL**

A process is a built-in class that allows one process to access and control another process once it has started. Users can declare variables of type process and safely pass them through tasks or incorporate them into other objects. The prototype for the process class is:

```
class process;
enum state { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED };
static function process self();
function state status();
task kill();
task await();
task suspend();
task resume();
endclass
```

Objects of type process are created internally when processes are spawned. Users cannot create objects of type process; attempts to call new shall not create a new process, and instead result in an error. The process class cannot be extended. Attempts to extend it shall result in a compilation error. Objects of type process are unique; they become available for reuse once the underlying process terminates and all references to the object are discarded. The self() function returns a handle to the current process, that is, a handle to the process making the call.

The status() function returns the process status, as defined by the state enumeration:

- ⌚ FINISHED Process terminated normally.
- ⌚ RUNNING Process is currently running (not in a blocking statement).
- ⌚ WAITING Process is waiting in a blocking statement.
- ⌚ SUSPENDED Process is stopped awaiting a resume.
- ⌚ KILLED Process was forcibly killed (via kill or disable).

### **Kill**

The kill() task terminates the given process and all its sub-processes, that is, processes spawned using fork statements by the process being killed. If the process to be terminated is not blocked waiting on some other condition, such as an event, wait expression, or a delay then the process shall be terminated at some unspecified time in the current time step.

## await

The await() task allows one process to wait for the completion of another process. It shall be an error to call this task on the current process, i.e., a process cannot wait for its own completion.

## suspend

The suspend() task allows a process to suspend either its own execution or that of another process. If the process to be suspended is not blocked waiting on some other condition, such as an event, wait expression, or a delay then the process shall be suspended at some unspecified time in the current time step. Calling this method more than once, on the same (suspended) process, has no effect.

## resume

The resume() task restarts a previously suspended process. Calling resume on a process that was suspended while blocked on another condition shall re-sensitize the process to the event expression, or wait for the wait condition to become true, or for the delay to expire. If the wait condition is now true or the original delay has transpired, the process is scheduled onto the Active or Reactive region, so as to continue its execution in the current time step. Calling resume on a process that suspends itself causes the process to continue to execute at the statement following the call to suspend.

The example below starts an arbitrary number of processes, as specified by the task argument N. Next, the task waits for the last process to start executing, and then waits for the first process to terminate. At that point the parent process forcibly terminates all forked processes that have not completed yet.

```
task do_n_way( int N );
    process job[1:N];
    for ( int j = 1; j <= N; j++ )
        fork
            automatic int k = j;
            begin job[j] = process::self(); ... ; end
            join_none
        for( int j = 1; j <= N; j++ ) // wait for all processes to start
            wait( job[j] != null );
            job[1].await(); // wait for first process to finish
        for ( int k = 1; k <= N; k++ ) begin
            if ( job[k].status != process::FINISHED )
                job[k].kill();
        end
    endtask
```

## INTERFACE

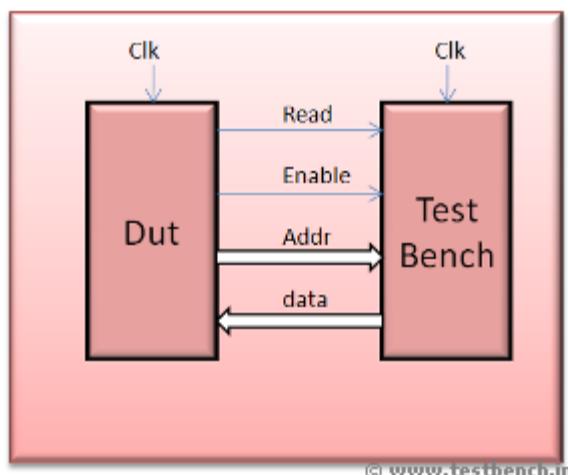
The communication between blocks of a digital system is a critical area. In Verilog, modules are connected using module ports. For large modules, this is not productive as it involves

- ⌚ Manually connecting hundreds of ports may lead to errors.
- ⌚ Detailed knowledge of all the port is required.
- ⌚ Difficult to change if the design changes.
- ⌚ More time consuming.
- ⌚ Most port declaration work is duplicated in many modules.

Let us see a verilog example:

```
module Dut (input clk, read, enable,  
           Input [7:0] addr,  
           output [7:0] data);  
  
....  
assign data = temp1 ? temp2 : temp3 ;  
always @(posedge clk)  
....  
endmodule  
module Testbench(input clk,  
                 Output read, enable,  
                 output [7:0] addr,  
                 input [7:0] data );  
endmodule
```

Integrating the above two modules in top module.



```

1 module top();
2   reg clk;
3   wire read, enable;
4   wire [7:0] addr;
5   wire [7:0] data;
6
7   Dut D (clk,read,enable,Addr,data);
8
9   Testbench TB(clk,read,enable,Addr,data);
10
11 endmodule

```

All the connection clk, read, enable, addr, data are done manually. Line 7 and 9 has same code structure which is duplicating work. If a new port is added, it needs changes in DUT ports, TestBench Ports, and in 7, 10 lines of Top module. This is time-consuming and maintaining it is complex as the port lists increases.

To resolve the above issues, SystemVerilog added a new powerful features called interface. Interface encapsulates the interconnection and communication between blocks.

Interface declaration for the above example:

```

interface intf #(parameter BW = 8)(input clk);
  logic read, enable;
  logic [BW -1 :0] addr,data;
endinterface :intf

```

Here the signals read, enable, addr, data are grouped in to "intf". Interfaces can have direction as input, output and inout also. In the above example, clk signal is used as input to interface. Interfaces can also have parameters like modules. Interface declaration is just like a module declaration. Uses keywords interface, endinterface for defining. Inside a module, use hierarchical names for signals in an interface.

TIP: Use wire type in case of multiple drivers. Use logic type in case of a single driver.

Let use see the DUT and Testbench modules using the above declared interface.

```

module Dut (intf dut_if); // declaring the interface
always @(posedge dut_if.clk)
  if(dut_if.read) // sampling the signal
    $display(" Read is asserted");

endmodule
module Testbench(intf tb_if);

initial
  begin
    tb_if.read = 0;
    repeat(3) #20 tb_if.read = ~tb_if.read;// driving a signal

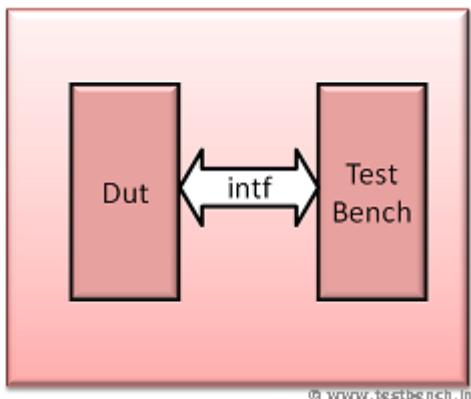
```

```

$finish;
end
endmodule

```

Integrating the above two modules in top module.



```

module top();
bit clk;
initial
    forever #5 clk = ~clk;
intf bus_if(clk); // interface instantiation
Dut d(bus_if); // use interface for connecting D and TB
Testbench TB (bus_if);
endmodule

```

See, how much code we got reduced in this small example itself. In the above example, I demonstrated the connectivity between DUT and TestBench. Interfaces can also be used for connectivity between the DUT sub modules also.

### Advantages Of Using Interface:

- An interface can be passed as single item.
- It allows structured information flow between blocks.
- It can contain anything that could be in a module except other module definitions or instance.
- Port definitions are independent from modules.
- Increases the reusability.
- It Interface can be declared in a separate file and can be compiled separately.
- Interfaces can contain tasks and functions; with this methods shared by all modules connecting to this information can be in one place.
- Interface can contain protocol checking using assertions and functional coverage blocks.
- Reduces errors which can cause during module connections.

⌚ Easy to add or remove a signal. Easy maintainability.

## **PORTS**

### **Interface Ports**

In the previous example, signal clk is declared as port to the interface. Interface Ports work similar to the module ports. Members of port list can be connected externally by name or position when the interface is instantiated as shown in line 3 of module top code.

### **Modports**

In the above example, we did not mention the direction of signals. The direction of the clk signal is input for both the Dut and Testbench modules. But for the rest of the signals, the direction is not same. To specify the direction of the signal w.r.t module which uses interface instead of port list, modports are used. Modport restrict interface access within a module based on the direction declared. Directions of signals are specified as seen from the module. In the modeport list, only signal names are used.

Let us see the modport usage with the previous example. 2 mod port definitions are needed, one for DUT and other for TestBench.

Interface declaration for the above example:

```
interface intf (input clk);
    logic read, enable,
    logic [7:0] addr,data;

    modport dut (input read,enable,addr,output data);
    modport tb (output read,enable,addr,input data);
endinterface :intf
```

In this example, the modport name selects the appropriate directional information for the interface signals accessed in the module header. Modport selection can be done in two ways. One at Module declaration , other at instantiation.

### **Modport Selection During Module Definition.**

```
module Dut (intf.dut dut_if); // declaring the interface with modport
    ....
    assign dut_if.data = temp1 ? temp2 : temp3; // using the signal in interface
    always @(posedge intf.clk)
    ....
endmodule
module Testbench(intf.tb tb_if);
    ....
    ....
endmodule

1 module top();
```

```

2 logic clk;
3 intf bus_if(clk); // interface instantiation
4 Dut d(bus_if); // Pass the interface
5 Testbench TB (Bus_if); // Pass the interface
6 endmodule

```

### Modport Selection During Module Instance.

```

module Dut (intf dut_if); // declaring the interface ....
  assign dut_if.data = temp1 ? temp2 : temp3 ; // using the signal in interface
  always @(posedge intf.clk)
  ....
endmodule
module Testbench(intf tb_if);
  ....
  ....
endmodule
1 module top();
2   logic clk;
3   intf bus_if(clk); // interface instantiation
4   Dut d(bus_if.dut); // Pass the modport into the module
5   Testbench TB (Bus_if.tb); // Pass the modport into the module
6 endmodule

```

A mod port can also define expressions. They can also define their own names. Module can use the modport declared name. For example

```

modport dut (input read,enable,.addr(2),output .d(data[1:5]));
  module dut(intf.dut dut_if);
    assign dut_if.d = temp; // using the signal name declared by modport.

```

## **INTERFACE METHODS**

### Methods In Interfaces

Interfaces can include task and function definitions. This allows a more abstract level of modeling.

```

interface intf (input clk);
  logic read, enable,
  logic [7:0] addr,data;
    task masterRead(input logic [7:0] raddr); // masterRead method
    ...
  endtask: masterRead
  task slaveRead; // slaveRead method
  ...
  endtask: slaveRead
endinterface :intf

```

## **CLOCKING BLOCK**

### **Clocking Blocks**

SystemVerilog adds the clocking block that identifies clock signals, and captures the timing and synchronization requirements of the blocks being modeled. A clocking block assembles signals that are synchronous to a particular clock, and makes their timing explicit. The clocking block is a key element in a cycle-based methodology, which enables users to write testbenches at a higher level of abstraction. Simulation is faster with cycle based methodology.

Depending on the environment, a testbench can contain one or more clocking blocks, each containing its own clock plus an arbitrary number of signals. These operations are as follows:

- ⌚ Synchronous events
- ⌚ Input sampling
- ⌚ Synchronous drives

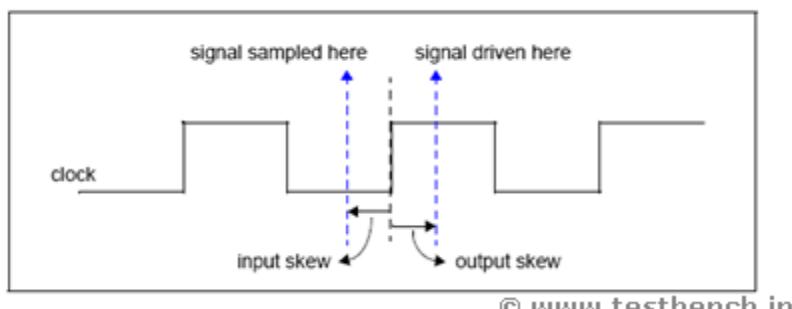
```
clocking cb @(posedge clk);  
default input #10ns output #2ns;
```

```
output read,enable,addr;  
input negedge data;  
endclocking
```

In the above example, the first line declares a clocking block called cb that is to be clocked on the positive edge of the signal clk. The second line specifies that by default all signals in the clocking block shall use a 10ns input skew and a 2ns output skew by default. The next line adds three output signals to the clocking block: read, enable and addr. The fourth line adds the signal data to the clocking block as input. Fourth line also contains negedge which overrides the skew ,so that data is sampled on the negedge of the clk.

### **Skew**

If an input skew is specified then the signal is sampled at skew time units before the clock event. If output skew is specified, then output (or inout) signals are driven skew time units after the corresponding clock event. A skew must be a constant expression, and can be specified as a parameter.



Skew can be specified in 3 ways.

⌚ #d : The skew is d time units. The time unit depends on the timescale of the block.

⌚ #dns : The skew is d nano seconds.

⌚ #1step : Sampling is done in the preponed region of current time stamp.

If skew is not specified, default input skew is 1step and output skew is 0.

Specifying a clocking block using a SystemVerilog interface can significantly reduce the amount of code needed to connect the TestBench without race condition. Clocking blocks add an extra level of signal hierarchy while accessing signals.

Interface declaration with clocking block:

```
interface intf (input clk);
    logic read, enable,
    logic [7:0] addr,data;
        clocking cb @(posedge clock); // clocking block for testbench
        default input #10ns output #2ns;
    output read,enable,addr;
        input data;
    endclocking
        modport dut (input read,enable,addr,output data);
        modport tb (clocking cb); // synchronous testbench modport
endinterface :intf
module testbench(intf.tb tb_if);
    .....
initial
    tb_if.cb.read <= 1; //writing to synchronous signal read
    ...
endmodule
```

### Cycle Delay

The ## operator can be used to delay execution by a specified number of clocking events, or clock cycles. What constitutes a cycle is determined by the default clocking in effect of module, interface, or program.

```
##<integer_expression>;
##3; // wait 3 cycles
##1 tb_if.addr <= 8'h00;// waits for 1 cycle and then writes address.
```

Using clocking blocks,cycle delays syntax gets reduced.

Instead of writing

```
repeat(3) @(posedge clock); sync_block.a <= 1;
```

Just use

```
##3 sync_block.a <= 1;
```

To schedule the assignment after 3 clocks, Just use,

```
sync_block.a <= ##3 1;  
To simply wait for 3 clock cycles,  
##3; can be used.
```

But there may be more than one clocking block is defined in a project.  
##3 waits for 3 clocks cycles,of the block which is defined as default.

```
default clocking sync_block @(posedge clock);
```

## VIRTUAL INTERFACE

### Virtual Interfaces

Virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up the design. A virtual interface allows the same subprogram to operate on different portions of a design, and to dynamically control the set of signals associated with the subprogram. Instead of referring to the actual set of signals directly, users are able to manipulate a set of virtual signals.

```
1 module testbench(intf.tb tb_if);  
2   virtual interface intf.tb local_if; // virtual interface.  
3   ....  
4   task read(virtual interface intf.tb l_if) // As argument to task  
5   ....  
6   initial  
7   begin  
8     Local_if = tb_if; // initializing virtual interface.  
9     Local_if.cb.read <= 1; //writing to synchronous signal read  
10    read(Local_if); // passing interface to task.  
11  end  
12 endmodule
```

In the above program, local\_if is just like a pointer. It represents an interface instance. Using keyword "virtual" , virtual interfaces instance is created. It does not have any signal. But it can hold physical interface. Tb\_if is the physical interface which is allocated during compilation time. You can drive and sample the signals in physical interface. In line 8, the physical interface tb\_if is assigned to local\_if. With this , we can drive and sample the physical signals. In line 9, read signal of tb\_if is accessed using local\_if.

### Advantages Of Virtual Interface

- 1) Virtual interface can be used to make the TestBench independent of the physical interface. It allows developing the test component independent of the DUT port while working with multi port protocol.
- 2) With virtual interface, we can change references to physical interface dynamically. Without virtual interfaces, all the connectivity is determined during compilation time, and therefore can't be randomized or reconfigured.
- 3) In multi port environment, it allows to access the physical interfaces using array index.

4) Physical interfaces are not allowed in object oriented programming, as physical interface is allocated at compilation time itself. Virtual interface which are set at run time allows to do object oriented programming with signals rather than just with variables.

5) Virtual interface variables can be passed as arguments to tasks, functions, or methods.

6) Allows to use Equality ( == ) and inequality ( != ) .

A virtual interface must be initialized before it can be used, by default, it points to null.

Attempting to use an uninitialized virtual interface will result in a run-time error.

### **Multi Bus Interface**

If you are working on protocol which has multiple sub bus protocol, there are several ways to create interfaces.

⌚ One Big single interface with all the sub protocol signals inside it. With single interface, it is easy to pass around the whole system. You have to name all the signals with the sub protocol prefix like pcie\_enable, Eth\_enable etc. Restrict the access using clocking blocks else all signals can be accessed by all the modules using this interface. Reusability will be very less as all the sub protocols are in one interface.

⌚ Using multiple interfaces, one for each sub protocol, each interface for each sub bus will increase the complexity while passing around the system. No need to prefix the sub protocol name as the sub protocol name is reflected in the interface name itself. With this you can only pass the sub interfaces required by other modules. All the interfaces will be reusable as each interface represents an individual protocol.

⌚ Using one big interface with sub multiple interfaces will be easy for passing around the system. With this the sub interfaces can be reused for other components and other designs.

### **SVTB N VERILOG DUT**

#### **Working With Verilog Dut:**

There are several ways to connect the Verilog DUT to SystemVerilog TestBench. Verilog DUT has port list where as SystemVerilog testbenchs uses interfaces. We will discuss 2 ways of connecting Verilog DUT to SystemVerilog TestBench.

#### **Connecting In Top:**

Verilog port list can be connected during DUT instantiation using interface hibachi signal names as shown in following code.

```
// DUT in Verilog
module Dut (input clk, read, enable,
    Input [7:0] addr,
    output [7:0] data);
    ...
    assign data = temp1 ? temp2 : temp3 ;
    always @(posedge clk)
    ...
endmodule
```

```

// SystemVerilog Code
// interface declaration with clocking block:
interface intf (input clk);
    logic read, enable;
    logic [7:0] addr,data;
endinterface
module testbench(intf.tb tb_if);
.....
endmodule
// integrating in top module.
module top();
    logic clk;
intf bus_if(clk); // interface instantiation
    Testbench TB (bus_if); // Pass the modport into the module
    Dut d(.clk(clk), // connect the verilog
        .read(bus_if.read), // RTL port using interface hierarchy signal name.
        .enable(bus_if.enable),
        .addr(bus_if.addr),
        .data(bus_if.data);
endmodule

```

### Connecting Using A Wrapper

We can also convert the verilog module with port list in to SystemVerilog module with interface by creating wrapper around the verilog module.

```

//wrapper for verilog DUT
module w_dut(intf wif);
    Dut d(.clk(wif.clk), // connect the verilog
        .read(wif.read), // RTL port using interface hierarchy signal name.
        .enable(wif.enable),
        .addr(wif.addr),
        .data(wif.data);
endmodule
//connecting the dut wrapper and testbench in top.
module top();
    logic clk;
intf bus_if(clk); // interface instantiation
    w_dut d(bus_if); // instance of dut wrapper
    Testbench TB (Bus_if);
endmodule

```

## **INTRODUCTION**

### **Brief Introduction To Oop**

Unlike procedural programming, here in the OOP programming model programs are organized around objects and data rather than actions and logic. Objects represent some concepts or things and like any other objects in the real Objects in programming language have certain behavior, properties, type, and identity. In OOP based language the principal aim is to find out the objects to manipulate and their relation between each other. OOP offers greater flexibility and compatibility than procedural language like verilog.

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your system, your desk, your chair.

Real-world objects share two characteristics: They all have state and behavior. System have state (name, color) and behavior (playing music, switch off). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

SystemVerilog is a object oriented programming and to understand the functionality of OOP in SystemVerilog, we first need to understand several fundamentals related to objects. These include class, method, inheritance, encapsulation, abstraction, polymorphism etc.

### **Class**

It is the central point of OOP and that contains data and codes with behavior. In SystemVerilog OOPS , everything happens within class and it describes a set of objects with common behavior. The class definition describes all the properties, behavior, and identity of objects present within that class.

### **Object**

Objects are the basic unit of object orientation with behavior, identity. As we mentioned above, these are part of a class but are not the same. An object is expressed by the variable and methods within the objects. Again these variables and methods are distinguished from each other as instant variables, instant methods and class variable and class methods.

### **Methods**

We know that a class can define both attributes and behaviors. Again attributes are defined by variables and behaviors are represented by methods. In other words, methods define the abilities of an object.

### **Inheritance**

This is the mechanism of organizing and structuring program. Though objects are distinguished from each other by some additional features but there are objects that share certain things common. In object oriented programming classes can inherit some common behavior and state from others. Inheritance in OOP allows to define a general class and later to organize some other classes simply adding some details with the old class definition. This saves work as the special

class inherits all the properties of the old general class and as a programmer you only require the new features. This helps in a better data analysis, accurate coding and reduces development time. In Verilog , to write a new definition using the existing definition is done inserting `ifdef compilation controls into the existing code.

### **Abstraction**

The process of abstraction in SystemVerilog is used to hide certain details and only show the essential features of the object. In other words, it deals with the outside view of an object.

### **Encapsulation**

This is an important programming concept that assists in separating an object's state from its behavior. This helps in hiding an object's data describing its state from any further modification by external component. In SystemVerilog there are three different terms used for hiding data constructs and these are public, private and protected . As we know an object can associate with data with predefined classes and in any application an object can know about the data it needs to know about. So any unnecessary data are not required by an object can be hidden by this process. It can also be termed as information hiding that prohibits outsiders in seeing the inside of an object in which abstraction is implemented.

### **Polymorphism**

It describes the ability of the object in belonging to different types with specific behavior of each type. So by using this, one object can be treated like another and in this way it can create and define multiple level of interface. Here the programmers need not have to know the exact type of object in advance and this is being implemented at runtime.

### **CLASS**

A class is an actual representation of an abstract data type. It therefore provides implementation details for the data structure used and operations.

#### **EXAMPLE:**

```
class A ;  
    // attributes:  
    int i  
    // methods:  
    task print  
endclass
```

**Definition (Class) :** A class is the implementation of an abstract data type . It defines attributes and methods which implement the data structure and operations of the abstract data type, respectively. Instances of classes are called objects. Consequently, classes define properties and behaviour of sets of objects.

In SV classes also contain Constraints for randomization control.

## **Class Properties:**

### **⌚ Instance Variables (Non-Static Fields) :**

Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the static keyword. Non-static fields are also known as instance variables because their values are unique to each instance of a class (to each object, in other words);

### **⌚ Class Variables (Static Fields) :**

A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.

### **⌚ Local Variables :**

Local Variables Similar to how an object stores its state in fields, a method will often store its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (for example, int count = 0;). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared , which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

### **⌚ Parameters :**

The important thing to remember is that parameters are always classified as "variables" not "fields".

### **⌚ Constants :**

Class properties can be made read-only by a const declaration like any other SystemVerilog variable.

## **OBJECT**

Objects are uniquely identifiable by a name. Therefore you could have two distinguishable objects with the same set of values. This is similar to traditional programming languages like verilog where you could have, say two integers i and j both of which equal to 2. Please notice the use of i and j in the last sentence to name the two integers. We refer to the set of values at a particular time as the state of the object.

### **EXAMPLE:**

```
class simple ;  
    int i;  
    int j;  
  
    task printf();  
        $display( i , j );
```

```

endtask
endclass
program main;
    initial
    begin
        simple obj_1;
        simple obj_2;
        obj_1 = new();
        obj_2 = new();
        obj_1.i = 2;
        obj_2.i = 4;
        obj_1.printf();
        obj_2.printf();
    end
endprogram

```

## RESULT

2	0
4	0

**Definition (Object)** An object is an instance of a class. It can be uniquely identified by its name and it defines a state which is represented by the values of its attributes at a particular time.

The state of the object changes according to the methods which are applied to it. We refer to these possible sequence of state changes as the behaviour of the object.

**Definition (Behaviour)** The behaviour of an object is defined by the set of methods which can be applied on it.

We now have two main concepts of object-orientation introduced, class and object. Object-oriented programming is therefore the implementation of abstract data types or, in more simple words, the writing of classes. At runtime instances of these classes, the objects, achieve the goal of the program by changing their states. Consequently, you can think of your running program as a collection of objects.

## Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class. In the following statements ,program creates an object and assigns it to a variable:

```

packet pkt = new(23, 94);
driver drvr = new(pkt,intf);

```

The first line creates an object of the packet class, and the second create an object of the driver class.

Each of these statements has three parts (discussed in detail below):

- ④ 1. Declaration: The code set in bold are all variable declarations that associate a variable

name with an object type.

- ⌚ 2. Instantiation: The new keyword is a SV operator that creates the object.
- ⌚ 3. Initialization: The new operator is followed by a call to a constructor, which initializes the new object.

### **Declaration:**

Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:

```
type name;
```

This notifies the compiler that you will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. You can also declare a reference variable on its own line. For example:

```
packet pkt;
```

If you declare pkt like this, its value will be undetermined until an object is actually created and assigned to it using the new method. Simply declaring a reference variable does not create an object. For that, you need to use the new operator. You must assign an object to pkt before you use it in your code. Otherwise, you will get a compiler error.

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, pkt, plus a reference pointing to nothing): . During simulation, the tool will not allocate memory for this object and error is reported. There will not be any compilation error.

### **EXAMPLE:**

```
class packet ;  
    int length = 0;  
    function new (int l);  
        length = l;  
    endfunction  
endclass  
  
program main;  
    initial  
    begin  
        packet pkt;  
        pkt.length = 10;  
    end  
endprogram
```

### **RESULT**

Error: null object access

### Instantiating A Class:

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```
packet pkt = new(10);
```

### Initializing An Object

Here's the code for the packet class:

#### **EXAMPLE**

```
class packet ;  
    int length = 0;  
    //constructor  
    function new (int l);  
        length = l;  
    endfunction  
endclass
```

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the packet class takes one integer arguments, as declared by the code (int l). The following statement provides 10 as value for those arguments:

```
packet pkt = new(10);
```

If a class does not explicitly declare any, the SV compiler automatically provides a no-argument constructor, called the default constructor. This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent.

### Constructor

SystemVerilog does not require the complex memory allocation and deallocation of C++.

Construction of an object is straightforward; and garbage collection, as in Java, is implicit and automatic. There can be no memory leaks or other subtle behavior that is so often the bane of C++ programmers.

The new operation is defined as a function with no return type, and like any other function, it must be nonblocking. Even though new does not specify a return type, the left-hand side of the assignment determines the return type.

### THIS

### Using The This Keyword

Within an instance method or a constructor, this is a reference to the current object , the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

For example, in the packet class was written like this

**EXAMPLE**

```
class packet ;  
    int length = 0;  
    //constructor  
    function new (int l);  
        length = l;  
    endfunction  
endclass
```

but it could have been written like this:

**EXAMPLE:**

```
class packet ;  
    int length = 0;  
    //constructor  
    function new (int length);  
        this.length = length;  
    endfunction  
endclass  
  
program main;  
initial  
begin  
    packet pkt;  
    pkt =new(10);  
    $display(pkt.length);  
end  
  
endprogram
```

**RESULT**

Each argument to the second constructor shadows one of the object's fields inside the constructor "length" is a local copy of the constructor's first argument. To refer to the length field inside the object , the constructor must use "this.length".

## **INHERITANCE**

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes.

**Definition (Inheritance)** Inheritance is the mechanism which allows a class A to inherit properties of a class B. We say ``A inherits from B''. Objects of class A thus have access to attributes and methods of class B without the need to redefine them. The following definition defines two terms with which we are able to refer to participating classes when they use inheritance.

**Definition (Superclass)** If class A inherits from class B, then B is called superclass of A.

**Definition (Subclass)** If class A inherits from class B, then A is called subclass of B.

Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behaviour as objects of the superclass. Superclasses are also called parent classes. Subclasses may also be called child classes or extended classes or just derived classes . Of course, you can again inherit from a subclass, making this class the superclass of the new subclass. This leads to a hierarchy of superclass/subclass relationships.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

### **What You Can Do In A Subclass:**

A subclass inherits all of the public and protected members of its parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members: The inherited fields can be used directly, just like any other fields.

☞ You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).

☞ You can declare new fields in the subclass that are not in the superclass.

- ⌚ The inherited methods can be used directly as they are.
- ⌚ You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- ⌚ You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- ⌚ You can declare new methods in the subclass that are not in the superclass.
- ⌚ You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

### Overriding

Following example shows, adding new variables, new methods, redefining existing methods.

#### EXAMPLE:

```

class parent;
  task printf();
    $display(" THIS IS PARENT CLASS ");
  endtask
endclass

class subclass extends parent;
  task printf();
    $display(" THIS IS SUBCLASS ");
  endtask
endclass

program main;

  initial
  begin
    parent p;
    subclass s;
    p = new();
    s = new();
    p.printf();
    s.printf();
  end

```

```
endprogram
```

## RESULT

THIS IS PARENT CLASS

THIS IS SUBCLASS

### Super

The super keyword is used from within a derived class to refer to members of the parent class. It is necessary to use super to access members of a parent class when those members are overridden by the derived class.

#### EXAMPLE:

```
class parent;  
  task printf();  
    $display(" THIS IS PARENT CLASS ");  
  endtask  
endclass
```

```
class subclass extends parent;
```

```
  task printf();  
    super.printf();  
  endtask  
endclass  
program main;  
  initial  
  begin  
    subclass s;  
    s = new();  
    s.printf();  
  end  
endprogram
```

## RESULT

THIS IS PARENT CLASS

The member can be a member declared a level up or be inherited by the class one level up. There is no way to reach higher (for example, super.super.count is not allowed).

Subclasses (or derived classes) are classes that are extensions of the current class whereas superclasses (parent classes or base classes) are classes from which the current class is extended, beginning with the original base class.

NOTE: When using the super within new, super.new shall be the first statement executed in the constructor. This is because the superclass must be initialized before the current class and, if the user code does not provide an initialization, the compiler shall insert a call to super.new automatically.

### Is Only Method

Programmers can override the existing code/functionality before existing code and replaces with new code as shown in below example.

#### EXAMPLE:

```
class parent;
    task printf();
        $display(" THIS IS PARENT CLASS ");
    endtask
endclass

class subclass extends parent;
    task printf();
        $display(" THIS IS SUBCLASS ");
    endtask
endclass

program main;

    initial
    begin
        subclass s;
        s = new();
        s.printf();
    end
endprogram
```

#### RESULT:

THIS IS SUBCLASS

### Is First Method

Programmers can add new lines of code/functionality before existing code as shown in below

example.

**EXAMPLE:**

```
class parent;
    task printf();
        $display(" THIS IS PARENT CLASS ");
    endtask
endclass

class subclass extends parent;
    task printf();
        $display(" THIS IS SUBCLASS ");
        super.printf();
    endtask
endclass

program main;

initial
begin
    subclass s;
    s = new();
    s.printf();
end
endprogram
```

**RESULT:**

```
THIS IS SUBCLASS
THIS IS PARENT CLASS
```

**Is Also Method**

Programmers can add new lines of code/functionality after the existing code as shown in below example.

**EXAMPLE:**

```
class parent;
    task printf();
        $display(" THIS IS PARENT CLASS ");
    endtask
endclass
```

```

class subclass extends parent;
  task printf();
    super.printf();
    $display(" THIS IS SUBCLASS ");
  endtask
endclass

program main;
  initial
  begin
    subclass s;
    s = new();
    s.printf();
  end
endprogram

```

## RESULT:

THIS IS PARENT CLASS  
 THIS IS SUBCLASS

### Overriding Constraints.

Programmers can override the existing constraint and replaces with new constraint as shown in below example.

#### EXAMPLE:

```

class Base;
  rand integer Var;
  constraint range { Var < 100 ; Var > 0 ;}
endclass

class Extended extends Base;
  constraint range { Var < 100 ; Var > 50 ;} // Overridding the Base class constraints.
endclass

program inhe_31;
  Extended obj;

  initial
  begin
    obj = new();
  end

```

```

for(int i=0 ; i < 100 ; i++)
    if(obj.randomize())
        $display(" Randomization sucessfull : Var = %0d ",obj.Var);
    else
        $display("Randomization failed");
end
endprogram

```

## RESULT:

Randomization sucessfull : Var = 77  
 Randomization sucessfull : Var = 86  
 Randomization sucessfull : Var = 76  
 Randomization sucessfull : Var = 89  
 Randomization sucessfull : Var = 86  
 Randomization sucessfull : Var = 76  
 Randomization sucessfull : Var = 96

### Overriding Datamembers

Only virtual methods truly override methods in base classes. All other methods and properties do not override but provide name hiding.

#### EXAMPLE

```

class base;
  int N = 3;

  function int get_N();
    return N;
  endfunction

endclass

class ext extends base;
  int N = 4;

  function int get_N();
    return N;
  endfunction

  function int get_N1();
    return super.N;
  endfunction

endclass

```

```

program main;
  initial
  begin
    ext e = new;
    base b = e;           // Note same object!
    $display(b.get_N()); // "3"
    $display(e.get_N()); // "4"
    $display(e.get_N1()); // "3" - super.N
  end
endprogram

```

## RESULT

3  
4  
3

### ENCAPSULATION

Encapsulation is a technique for minimizing interdependencies among modules by defining a strict external communication. This way, internal coding can be changed without affecting the communication, so long as the new implementation supports the same (or upwards compatible) external communication.

Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effects.

The implementation of an object can be changed without affecting the application that uses it for:

Improving performance, fix a bug, consolidate code or for porting.

#### Access Specifiers:

In SystemVerilog, unqualified class properties and methods are public, available to anyone who has access to the object's name.

A member identified as local is available only to methods inside the class. Further, these local members are not visible within subclasses. Of course, nonlocal methods that access local class properties or methods can be inherited and work properly as methods of the subclass.

#### **EXAMPLE: local variblg error**

```

class base;
  local int i;

```

```

endclass
program main;
initial
begin
    base b = new();
    b.i = 123;
end
endprogram

```

### **RESULT:**

Local member 'i' of class 'base' is not accessible from scope 'main'

The above examples gives compilation error.

### **EXAMPLE: local varible access using method**

```

class base;
    local int i;

    task set(int j);
        i = j;
        $display(i);
    endtask
endclass

program main;
    initial
    begin
        base b = new();
        b.set(123);
    end
endprogram

```

### **RESULT**

123

### **EXAMPLE: local varible access in subclass**

```

class base;

```

```

local int i;
endclass
class ext extends base;
    function new();
        i = 10;
    endfunction
endclass

```

## RESULT

Local member 'i' of class 'base' is not accessible from scope 'ext'

A protected class property or method has all of the characteristics of a local member, except that it can be inherited; it is visible to subclasses.

### EXAMPLE: protected varible

```

class base;
    protected int i;
endclass

```

```

class ext extends base;
    function new();
        i = 10;
    endfunction
endclass

```

### EXAMPLE: protected varible in 2 level of inheritance

```

class base;
    local int i;
endclass
class ext extends base;
    protected int i;
endclass
class ext2 extends ext;
    function new();
        i = 10;
    endfunction
endclass

```

In the above example, the varible i is overloaded in subclass with different qualifier.

### EXAMPLE: Error access to protected varible.

```
class base;
```

```

protected int i;
endclass
program main;
  initial
  begin
    base b = new();
    b.i = 123;
  end
endprogram

```

## RESULT

Protected member 'i' of class 'base' is not accessible from scope 'main'

Within a class, a local method or class property of the same class can be referenced, even if it is in a different instance of the same class.

## EXAMPLE

```

class Packet;
  local integer i;

  function integer compare (Packet other);
    compare = (this.i == other.i);
  endfunction
endclass

```

A strict interpretation of encapsulation might say that other.i should not be visible inside of this packet because it is a local class property being referenced from outside its instance. Within the same class, however, these references are allowed. In this case, this.i shall be compared to other.i and the result of the logical comparison returned.

## POLYMORPHISM

Polymorphism allows an entity to take a variety of representations. Polymorphism means the ability to request that the same Operations be performed by a wide range of different types of things. Effectively, this means that you can ask many different objects to perform the same action. Override polymorphism is an override of existing code. Subclasses of existing classes are given a "replacement method" for methods in the superclass. Superclass objects may also use the replacement methods when dealing with objects of the subtype. The replacement method that a subclass provides has exactly the same signature as the original method in the superclass.

Polymorphism allows the redefining of methods for derived classes while enforcing a common interface. To achieve polymorphism the 'virtual' identifier must be used when defining the base

class and method(s) within that class.

**EXAMPLE: without virtual**

```
class A ;  
    task disp ();  
        $display(" This is class A ");  
    endtask  
endclass  
  
class EA extends A ;  
    task disp ();  
        $display(" This is Extended class A ");  
    endtask  
endclass  
  
program main ;  
    EA my_ea;  
    A my_a;  
  
    initial  
    begin  
        my_a = new();  
        my_a.disp();  
  
        my_ea = new();  
        my_a = my_ea;  
        my_a.disp();  
    end  
endprogram
```

**RESULTS**

This is class A

This is class A

**EXAMPLE: with virtual**

```
class A ;  
    virtual task disp ();  
        $display(" This is class A ");  
    endtask  
endclass
```

```

class EA extends A ;
  task disp ();
    $display(" This is Extended class A ");
  endtask
endclass

program main ;
  EA my_ea;
  A my_a;

  initial
  begin
    my_a = new();
    my_a.disp();

    my_ea = new();
    my_a = my_ea;
    my_a.disp();
  end
endprogram

```

## RESULTS

This is class A

This is Extended class A

Observe the above two outputs. Methods which are declared as virtual are executing the code in the object which is created.

The methods which are added in the subclasses which are not in the parent class cannot be accessed using the parent class handle. This will result in a compilation error. The compiler checks whether the method is existing in the parent class definition or not.

## EXAMPLE:

```

class A ;
endclass

class EA extends A ;
  task disp ();
    $display(" This is Extended class A ");
  endtask
endclass

program main ;
  EA my_ea;

```

```

A my_a;

initial
begin
    my_ea = new();
    my_a = my_ea;
    my_ea.disp();
    my_a.disp();
end
endprogram

```

### **RESULT:**

Member disp not found in class A

To access the variable or method which are only in the subclass and not in the parent class, revert back the object to the subclass handle.

### **EXAMPLE:**

```

class A ;
endclass

```

```

class EA extends A ;
    task disp ();
        $display(" This is Extended class A ");
    endtask
endclass

```

```

program main ;
    EA my_ea;
    A my_a;

```

```

initial
begin
    my_ea = new();
    my_a = my_ea;
    just(my_a);
end
endprogram

```

```

task just(A my_a);
    EA loc;
    $cast(loc,my_a);

```

```
loc.disp();  
endtask
```

## RESULT

This is Extended class A

Let us see one more example, A parent class is extended and virtual method is redefined in the subclass as non virtual method. Now if further extention is done to the class, then the method is still considered as virtual method and Polymorphism can be achieved still. It is advised to declare a method as virtual in all its subclass, if it is declared as virtual in baseclass , to avoid confusion to the end user who is extend the class.

## EXAMPLE:

```
class A ;  
    virtual task disp ();  
        $display(" This is class A ");  
    endtask  
endclass  
  
class EA_1 extends A ;  
    task disp ();  
        $display(" This is Extended 1 class A ");  
    endtask  
endclass  
  
class EA_2 extends EA_1 ;  
    task disp ();  
        $display(" This is Extended 2 class A ");  
    endtask  
endclass  
  
program main ;  
    EA_2 my_ea;  
    EA_1 my_a;  
  
    initial  
    begin  
        my_ea = new();  
        my_a = my_ea;  
        my_a.disp();  
        just(my_a);
```

```

end
endprogram

task just(A my_a);
  EA_1 loc;
  $cast(loc,my_a);
  loc.disp();
endtask

```

## RESULT

This is Extended 2 class A  
 This is Extended 2 class A

### **ABSTRACT CLASSES**

With inheritance we are able to force a subclass to offer the same properties like their superclasses. Consequently, objects of a subclass behave like objects of their superclasses.

Sometimes it make sense to only describe the properties of a set of objects without knowing the actual behaviour beforehand

Abstract classes are those which can be used for creation of handles. However their methods and constructors can be used by the child or extended class. The need for abstract classes is that you can generalize the super class from which child classes can share its methods. The subclass of an abstract class which can create an object is called as "concrete class".

### EXAMPLE: using abstract class

```

virtual class A ;
  virtual task disp ();
    $display(" This is class A ");
  endtask
endclass

class EA extends A ;
  task disp ();
    $display(" This is Extended class A ");
  endtask
endclass

program main ;
  EA my_ea;

```

```

A my_a;

initial
begin
    my_ea = new();
    my_a = my_ea;
    my_ea.disp();
    my_a.disp();
end
endprogram

```

## RESULT

This is Extended class A

This is Extended class A

**EXAMPLE: creating object of virtual class**

```

virtual class A ;
    virtual task disp ();
        $display(" This is class A ");
    endtask
endclass

```

```

program main ;

```

```

    A my_a;

```

```

initial
begin
    my_a = new();
    my_a.disp();
end
endprogram

```

## RESULT

Abstract class A cannot be instantiated

Virtual keyword is used to express the fact that derived classes must redefine the properties to fulfill the desired functionality. Thus from the abstract class point of view, the properties are only specified but not fully defined. The full definition including the semantics of the properties must be provided by derived classes.

Definition (Abstract Class) A class A is called abstract class if it is only used as a superclass for

other classes. Class A only specifies properties. It is not used to create objects. Derived classes must define the properties of A.

## **PARAMETERISED CLASS**

### **Type Parameterised Class**

At the time, when we write down a class definition, we must be able to say that this class should define a generic type. However, if we don't know with which types the class will be used. Consequently, we must be able to define the class with help of a placeholder to which we refer as if it is the type on which the class operates. Thus, the class definition provides us with a template of an actual class. The actual class definition is created once we declare a particular object. Let's illustrate this with the following example. Suppose, you want to define a list class which should be a generic type. Thus, it should be possible to declare list objects for integers, bits, objects or any other type.

### **EXAMPLE**

```
class List #(type T = int);
    //attributes:
    T data_node;
    .....
    .....
    // methods:
    task append(T element);
    function T getFirst();
    function T getNext();
    .....
    .....
endclass
```

The above template class List looks like any other class definition. However, the first line declares List to be a template for various types. The identifier T is used as a placeholder for an actual type. For example, append() takes one element as an argument. The type of this element will be the data type with which an actual list object is created. For example, we can declare a list object for "packet" if a definition for the type "packet" exists:

### **EXAMPLE**

```
List#(packet) pl; // Object pl is a list of packet
List#(string) sl; // Object sl is a list of strings
```

The first line declares List#(packet) to be a list for packets. At this time, the compiler uses the template definition, substitutes every occurrence of T with "packet" and creates an actual class definition for it. This leads to a class definition similar to the one that follows:

## EXAMPLE

```
class List {  
    //attributes:  
    packet data_node;  
    .....  
    .....  
    // methods:  
    task append(packet element);  
    function packet getFirst();  
    function packet getNext();  
    .....  
    .....  
}
```

This is not exactly, what the compiler generates. The compiler must ensure that we can create multiple lists for different types at any time. For example, if we need another list for, say "strings", we can write:

## EXAMPLE

```
List#(packet) pl; // Object pl is a list of packet  
List#(string) sl; // Object sl is a list of strings
```

In both cases the compiler generates an actual class definition. The reason why both do not conflict by their name is that the compiler generates unique names. However, since this is not viewable to us, we don't go in more detail here. In any case, if you declare just another list of "strings", the compiler can figure out if there already is an actual class definition and use it or if it has to be created. Thus,

## EXAMPLE

```
List#(packet) rcv_pkt; // Object rcv_pkt is a list of packet  
List#(packet) sent_pkt; // Object sent_pkt is a list of packet
```

will create the actual class definition for packet List and will reuse it for another List. Consequently, both are of the same type. We summarize this in the following definition:

**Definition (Parameterized Class)** If a class A is parameterized with a data type B, A is called template class. Once an object of A is created, B is replaced by an actual data type. This allows the definition of an actual class based on the template specified for A and the actual data type.

We are able to define template classes with more than one parameter. For example, directories

are collections of objects where each object can be referenced by a key. Of course, a directory should be able to store any type of object. But there are also various possibilities for keys. For instance, they might be strings or numbers. Consequently, we would define a template class Directory which is based on two type parameters, one for the key and one for the stored objects.

### **Value Parameterised Class**

The normal Verilog parameter mechanism is also used to parameterize a class.

#### **EXAMPLE**

```
class Base#(int size = 3);
    bit [size:0] a;

    task disp();
        $display(" Size of the vector a is %d ",$size(a));
    endtask
endclass

program main();
    initial
    begin
        Base B1;
        Base#(4) B2;
        Base#(5) B3;
        B1 = new();
        B2 = new();
        B3 = new();
        B1.disp();
        B2.disp();
        B3.disp();
    end
endprogram
```

#### **RESULT**

Size of the vector a is	4
Size of the vector a is	5
Size of the vector a is	6

Instances of this class can then be instantiated like modules or interfaces:

### EXAMPLE

```
vector #(10) vten; // object with vector of size 10
vector #(.size(2)) vtwo; // object with vector of size 2
typedef vector#(4) Vfour; // Class with vector of size 4
```

### Generic Parameterised Class

A specialization is the combination of a specific generic class with a unique set of parameters. Two sets of parameters shall be unique unless all parameters are the same as defined by the following rules:

- ⌚ a) A parameter is a type parameter and the two types are matching types.
- ⌚ b) A parameter is a value parameter and both their type and their value are the same.

All matching specializations of a particular generic class shall represent the same type. The set of matching specializations of a generic class is defined by the context of the class declaration. Because generic classes in a package are visible throughout the system, all matching specializations of a package generic class are the same type. In other contexts, such as modules or programs, each instance of the scope containing the generic class declaration creates a unique generic class, thus, defining a new set of matching specializations.

A generic class is not a type; only a concrete specialization represents a type. In the example above, the class vector becomes a concrete type only when it has had parameters applied to it, for example:

```
typedef vector my_vector; // use default size of 1
```

### EXAMPLE

```
vector#(6) vx; // use size 6
```

To avoid having to repeat the specialization either in the declaration or to create parameters of that type, a **typedef** should be used:

### EXAMPLE

```
typedef vector#(4) Vfour;
typedef stack#(Vfour) Stack4;
Stack4 s1, s2; // declare objects of type Stack4
```

### Extending Parameterised Class

A parameterized class can extend another parameterized class. For example:

## EXAMPLE

```
class C #(type T = bit);
...
endclass // base class

class D1 #(type P = real) extends C; // T is bit (the default)
```

Class D1 extends the base class C using the base class's default type (bit) parameter.

```
class D2 #(type P = real) extends C #(integer); // T is integer
```

Class D2 extends the base class C using an integer parameter.

```
class D3 #(type P = real) extends C #(P); // T is P
```

Class D3 extends the base class C using the parameterized type (P) with which the extended class is parameterized.

## NESTED CLASSES

A nested class is a class whose definition appears inside the definition of another class, as if it were a member of the other class. The SystemVerilog programming language allows you to define a class within another class.

## EXAMPLE

```
class StringList;

    class Node; // Nested class for a node in a linked list.
        string name;
        Node link;
    endclass

    endclass

    class StringTree;
```

class Node; // Nested class for a node in a binary tree.

```

string name;
Node left, right;
endclass

endclass
// StringList::Node is different from StringTree::Node

```

Nesting allows hiding of local names and local allocation of resources. This is often desirable when a new type is needed as part of the implementation of a class. Declaring types within a class helps prevent name collisions and the cluttering of the outer scope with symbols that are used only by that class. Type declarations nested inside a class scope are public and can be accessed outside the class

### Why Use Nested Classes

There are several compelling reasons for using nested classes, among them:

- ⌚ It is a way of logically grouping classes that are only used in one place.
- ⌚ It increases encapsulation.
- ⌚ Nested classes can lead to more readable and maintainable code.

Logical grouping of classes : If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation : Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code:Nesting small classes within top-level classes places the code closer to where it is used.

### CONSTANT

SystemVerilog adds another form of a local constant, const. A const form of constant differs from a localparam constant in that the localparam must be set during elaboration, whereas a const can be set during simulation, such as in an automatic task.

### Constant Class

An instance of a class (an object handle) can also be declared with the const keyword.

## EXAMPLE

```
const class_name object = new(3,3,4);
```

In other words, the object acts like a variable that cannot be written. The arguments to the new method must be constant expressions. The members of the object can be written (except for those members that are declared const).

## Global Constant

Global constant class properties include an initial value as part of their declaration. They are similar to other const variables in that they cannot be assigned a value anywhere other than in the declaration.

## EXAMPLE

```
class Jumbo_Packet;  
  const int max_size = 9 * 1024; // global constant  
  byte payload [];  
  
  function new( int size );  
    payload = new[ size > max_size ? max_size : size ];  
  endfunction  
endclass
```

## EXAMPLE: error : chaning const variable

```
class A ;  
  const int i= 10;  
endclass  
  
program main ;  
  A my_a;
```

```
  initial  
  begin  
    my_a = new();  
    my_a.i = 55;  
  end  
endprogram
```

## RESULT

Error : Variable 'i' declared as 'const' .

## Instance Constants

Instance constants do not include an initial value in their declaration, only the const qualifier. This type of constant can be assigned a value at run time, but the assignment can only be done once in the corresponding class constructor not from outside or any other method.

### EXAMPLE

```
class A ;  
    const int i;  
  
    function new();  
        i = 20;  
    endfunction  
endclass  
  
program main ;  
    A my_a;  
  
    initial  
    begin  
        my_a = new();  
        $display(my_a.i);  
    end  
endprogram
```

### RESULT

20

EXAMPLE: error : assignment done twice

```
class A ;  
    const int i;  
  
    function new();  
        i = 20;  
        i++;  
    endfunction  
endclass
```

### RESULT

Instance constants assignment can only be done once

**EXAMPLE: error : assignment in other method**

```
class A ;  
  const int i;  
  
  task set();  
    i = 20;  
  endtask  
endclass
```

## RESULT

Instance constants assignment allowed only in class constructor.

Typically, global constants are also declared static because they are the same for all instances of the class. However, an instance constant cannot be declared static because that would disallow all assignments in the constructor

## STATIC

### Static Class Properties

A static property is a class variable that is associated with the class, rather than with an instance of the class (a.k.a., an object). This means that when it is changed, its change is reflected in all instances of the class. Static properties are declared with the static keyword. If you need to access a static property inside a class, you can also use the magic keywords "this" and "super", which resolve to the current class and the parent of the current class, respectively. Using "this" and "super" allows you to avoid having to explicitly reference the class by name.

**EXAMPLE: Using object name**

```
class A ;  
  static int i;  
endclass  
  
program main ;  
  A obj_1;  
  A obj_2;  
  
  initial  
  begin  
    obj_1 = new();  
    obj_2 = new();  
    obj_1.i = 123;  
    $display(obj_2.i);  
  end  
endprogram
```

**RESULT** 123

The static class properties can be accessed using class name.

**EXAMPLE: using class name**

```
class A ;  
    static int i;  
endclass  
  
program main ;  
    A obj_1;  
  
    initial  
    begin  
        obj_1 = new();  
        obj_1.i = 123;  
        $display(A::i);  
    end  
endprogram
```

**RESULT**

123

The static class properties can be used without creating an object of that type.

**EXAMPLE: without creating object**

```
class A ;  
    static int i;  
endclass  
  
program main ;  
    A obj_1;  
    A obj_2;  
  
    initial  
    begin  
        obj_1.i = 123;  
        $display(obj_2.i);  
    end  
endprogram
```

**RESULT**

123

**EXAMPLE:** using the object name, without creating object

```
class A ;  
  static int i;  
endclass  
  
program main ;  
  A obj_1;  
  
  initial  
  begin  
    obj_1.i = 123;  
    $display(A::i);  
  end  
endprogram
```

## RESULT

123

### Static Methods

Methods can be declared as static. A static method is subject to all the class scoping and access rules, but behaves like a regular subroutine that can be called outside the class.

**EXAMPLE**

```
class A ;  
  static task incr();  
    int j; //automatic variable  
    j++;  
    $display("J is %d",j);  
  endtask  
endclass  
  
program main ;  
  A obj_1;  
  A obj_2;  
  
  initial  
  begin  
    $display("Static task - static task with automatic variables");  
    obj_1 = new();  
    obj_2 = new();
```

```

obj_1.incr();
obj_2.incr();
obj_1.incr();
obj_2.incr();
obj_1.incr();

$display("Static task - Each call to task will create a separate copy of 'j' and increment it");

end
endprogram

```

## RESULT

Static task - static task with automatic variables

```

J is      1

```

Static task - Each call to task will create a separate copy of 'j' and increment it

A static method has no access to nonstatic members (class properties or methods), but it can directly access static class properties or call static methods of the same class. Access to nonstatic members or to the special this handle within the body of a static method is illegal and results in a compiler error.

## EXAMPLE

```

class A ;
  int j;

  static task incr();
    j++;
    $display("J is %d",j);
  endtask
endclass

```

```
program main ;
```

```
  A obj_1;
  A obj_2;
```

```
initial
```

```
begin
```

```
  obj_1 = new();
  obj_2 = new();
  obj_1.incr();
```

```
    obj_2.incr();
end
endprogram
```

## RESULT

A static method has no access to nonstatic members (class properties or methods).  
Static methods cannot be virtual.

## EXAMPLE

```
class A ;
  int j;

  virtual static task incr();
    $display("J is %d",j);
  endtask
endclass
```

## RESULT

Error : Static methods cannot be virtual.  
The static methods can be accessed using class name.

## EXAMPLE: using class name

```
class A ;
  static task who();
    $display(" Im static method ");
  endtask
endclass

program main;
  initial
    A.who();
endprogram
```

## RESULT

Im static method.  
The static methods can be used without creating an object of that type.

## EXAMPLE: without creating object

```
class A ;
  static task who();
    $display(" Im static method ");
```

```

endtask
endclass

program main;
  A a;

  initial
    a.who();
endprogram

```

## RESULT

Im static method.

### Static Lifetime Method.

By default, class methods have automatic lifetime for their arguments and variables.

All variables of a static lifetime method shall be static in that there shall be a single variable corresponding to each declared local variable in a class , regardless of the number of concurrent activations of the method.

### EXAMPLE

```

class A ;
  task static incr();
    int i; //static variable
    $display(" i is %d ",i);
    i++;
endtask
endclass

```

```

program main;
  A a;
  A b;
  initial
    begin
      $display("Static lifetime - non static task with static variables");
      a = new();
      b = new();
      a.incr();
      b.incr();
      a.incr();
      b.incr();
      a.incr();
    end

```

```

$display("Static lifetime - Each call to task will use a single value of 'j' and increment it");
end
endprogram

```

## **RESULT**

Static lifetime - non static task with static variables

```

i is      0
i is      1
i is      2
i is      3
i is      4

```

Static lifetime - Each call to task will use a single value of 'j' and increment it

Verilog-2001 allows tasks to be declared as automatic, so that all formal arguments and local variables are stored on the stack. SystemVerilog extends this capability by allowing specific formal arguments and local variables to be declared as automatic within a static task, or by declaring specific formal arguments and local variables as static within an automatic task.

By default, class methods have automatic lifetime for their arguments and variables.

## **EXAMPLE**

```

class packet;
  static int id;
  //----static task using automatic fields ---//
  static task packet_id();
    int count; // by default static task fields are automatic
    id=count; // writing in to static variable
    $display("id=%d count=%d",id,count);
    count++;
  endtask

  function new();
    int pckt_type;
    pckt_type++;
    $display("pckt_type=%d",pckt_type);
  endfunction

endclass

program stic_1;
  packet jumbo,pause,taggd;
initial
  begin
    jumbo=new();
    jumbo.packet_id();
  end

```

```
pause=new();
pause.packt_id();
taggd=new();
taggd.packt_id();
end
```

**endprogram**

## RESULTS

```
pckt_type= 1; id= 0; count=      0
pckt_type= 1; id= 0 ; count=      0
pckt_type= 1; id= 0; count=      0
```

SystemVerilog allows specific formal arguments and local variables to be declared as automatic within a static task, or by declaring specific formal arguments and local variables as static within an automatic task.

## EXAMPLE

**class** packet;

```
static int id,pckt_type;
//---static task with static field----/
static task packt_id();
static int count; //explicit declaration of fields as static
id=count; //writing in to static variable
$display("id=%d count=%d",id,count);
count++;
endtask
```

```
function new();
pckt_type++;
$display("pckt_type=%d",pckt_type);
endfunction
```

**endclass**

```
program stic_2;
packet jumbo,pause,taggd;
```

```
initial
begin
jumbo=new();
jumbo.packt_id();
pause=new();
pause.packt_id();
```

```
    taggd=new();
    taggd.packt_id();
end
```

### **endprogram**

#### **RESULTS**

```
pcpt_type= 1; id= 0 count= 0;
pcpt_type= 2; id= 1 count= 1;
pcpt_type= 3 ; id= 2 count= 2;
```

### **CASTING**

It is always legal to assign a subclass variable to a variable of a class higher in the inheritance tree.

#### **EXAMPLE**

```
class parent;
    int i = 10;
endclass
```

```
class subclass extends parent;
    int j;
```

```
function new();
    j = super.i;
    $display("J is %d",j);
endfunction
endclass
```

```
program main;
    initial
    begin
        subclass s;
        s = new();
    end
endprogram
```

#### **RESULT**

J is 10

It is never legal to directly assign a superclass variable to a variable of one of its subclasses.

However, it is legal to assign a superclass handle to a subclass variable if the superclass handle refers to an object of the given subclass.

SystemVerilog provides the \$cast system task to assign values to variables that might not ordinarily be valid because of differing data type. To check whether the assignment is legal, the dynamic cast function \$cast() is used . The syntax for \$cast() is as follows:

```
task $cast( singular dest_handle, singular source_handle );  
or  
function int $cast( singular dest_handle, singular source_handle );
```

When called as a task, \$cast attempts to assign the source expression to the destination variable. If the assignment is invalid, a run-time error occurs, and the destination variable is left unchanged.

#### EXAMPLE : \$cast as task

```
class B;  
    virtual task print();  
        $display(" CLASS B ");  
    endtask  
endclass  
  
class E_1 extends B;  
    virtual task print();  
        $display(" CLASS E_1 ");  
    endtask  
endclass  
  
class E_2 extends B;  
    virtual task print();  
        $display(" CLASS E_2 ");  
    endtask  
endclass  
  
program main;  
    initial  
    begin  
        B b;  
        E_1 e1;  
        E_2 e2;  
  
        e1 = new();  
        $cast(b,e1);  
        b.print();
```

```

    end
endprogram
RESULT
CLASS E_1
EXAMPLE : $cast as task with error
class B;
    virtual task print();
        $display(" CLASS B ");
    endtask
endclass

class E_1 extends B;
    virtual task print();
        $display(" CLASS E_1 ");
    endtask
endclass

class E_2 extends B;
    virtual task print();
        $display(" CLASS E_2 ");
    endtask
endclass

program main;
    initial
    begin
        B b;
        E_1 e1;
        E_2 e2;

        e1 = new();
        $cast(e2,e1);

    end
endprogram

```

RESULT

Error: Dynamic cast failed

When called as a function, \$cast attempts to assign the source expression to the destination variable and returns 1 if the cast is legal. If the cast fails, the function does not make the assignment and returns 0. When called as a function, no run-time error occurs, and the destination variable is left unchanged. It is important to note that \$cast performs a run-time check. No type checking is done by the compiler, except to check that the destination variable and source expression are singulars.

#### EXAMPLE : \$cast as function

```
class B;
    virtual task print();
        $display(" CLASS B ");
    endtask
endclass

class E_1 extends B;
    virtual task print();
        $display(" CLASS E_1 ");
    endtask
endclass

class E_2 extends B;
    virtual task print();
        $display(" CLASS E_2 ");
    endtask
endclass

program main;
    initial
    begin
        B b;
        E_1 e1;
        E_2 e2;

        e1 = new();
        //calling $cast like a task
        //Return value is not considered
        $cast(b,e1);
        which_class(b);

        e2 = new();
```

```

//calling $cast like a task
//Return value is not considered
    $cast(b,e2);
    which_class(b);
end
endprogram

task which_class(B b);
    E_1 e1;
    E_2 e2;
//calling $cast like a function
//Return value is considered for action
if($cast(e1,b))
    $display(" CLASS E_1 ");
if($cast(e2,b))
    $display(" CLASS E_2 ");
endtask

```

## RESULT

CLASS E\_1  
CLASS E\_2

When used with object handles, \$cast() checks the hierarchy tree (super and subclasses) of the source\_expr to see whether it contains the class of dest\_handle. If it does, \$cast() does the assignment. Otherwise, the error is issued.

Assignment of Extended class object to Base class object is allowed. It is Illegal to assign Base class object to Extended class.

## EXAMPLE

```

class Base;
endclass
class Exten extends Base;
endclass

```

```

program main;

initial
begin
    Base B;
    Exten E;
    B = new();
    if(!$cast(E,B))

```

```

$display(" Base class object B canot be assigned to Extended class Handle.");
// Deallocate object B
B = null;
E = new();
if(!$cast(B,E))
    $display(" Extended class object E canot be assigned to Base class Handle.");
end
endprogram

```

## RESULT

Base class object B canot be assigned to Extended class Handle.

## COPY

The terms "deep copy" and "shallow copy" refer to the way objects are copied, for example, during the invocation of a copy constructor or assignment operator.

### EXAMPLE:

```

class B;
    int i;
endclass

program main;
initial
begin
    B b1;
    B b2;
    b1 = new();
    b1.i = 123;
    b2 = b1;
    $display( b2.i );
end
endprogram

```

### RESULTS:

123

In the above example, both objects are pointing to same memory. The properties did not get copied. Only the handle is copied.

## Shallow Copy

A shallow copy of an object copies all of the member field values.

### EXAMPLE:

```

class B;
    int i;
endclass

```

```

program main;
  initial
  begin
    B b1;
    B b2;
    b1 = new();
    b1.i = 123;
    b2 = new b1;
    b2.i = 321;
    $display( b1.i );
    $display( b2.i );
  end
endprogram

```

**RESULTS:**

123

321

This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied, but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The assignment operator make shallow copies.

**EXAMPLE:**

```

class A;
  int i;
endclass

```

```

class B;
  A a;
endclass

```

```

program main;
  initial
  begin
    B b1;
    B b2;
    b1 = new();
    b1.a = new();
    b1.a.i = 123;
    b2 = new b1;
    $display( b1.a.i );
  end
endprogram

```

```

$display( b2.a.i );
b1.a.i = 321;
$display( b1.a.i );
$display( b2.a.i );
end
endprogram

```

## RESULT

```

123
123
321
321

```

In the above example, the variable i is changed to which is inside the object of . This change is seen because both the objects are pointing to same memory location.

### Deep Copy

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disastrous consequences.

#### EXAMPLE:

```

class A;
  int i;
endclass

class B;
  A a;

task copy(A a);
  this.a = new a;
endtask

endclass

program main;
  initial
  begin
    B b1;
    B b2;
    b1 = new();
    b1.a = new();
    b1.a.i = 123;
  end
endprogram

```

```
b2 = new b1;  
b2.copy(b1.a);  
$display( b1.a.i );  
$display( b2.a.i );  
b1.a.i = 321;  
$display( b1.a.i );  
$display( b2.a.i );
```

**end**

**endprogram**

**RESULTS:**

```
123  
123  
321  
123
```

### **Clone**

A clone method returns a new object whose initial state is a copy of the current state of the object on which clone was invoked. Subsequent changes to the clone will not affect the state of the original. Copying is usually performed by a clone() method of a class which is user defined. This method usually, in turn, calls the clone() method of its parent class to obtain a copy, and then does any custom copying procedures. Eventually this gets to the clone() method of Object (the uppermost class), which creates a new instance of the same class as the object and copies all the fields to the new instance (a "shallow copy"). After obtaining a copy from the parent class, a class's own clone() method may then provide custom cloning capability, like deep copying (i.e. duplicate some of the structures referred to by the object) .

One disadvantage is that the return type of clone() is Object, and needs to be explicitly cast back into the appropriate type (technically a custom clone() method could return another type of object; but that is generally inadvisable).

One advantage of using clone() is that since it is an overridable method, we can call clone() on any object, and it will use the clone() method of its actual class, without the calling code needing to know what that class is (which would be necessary with a copy constructor).

**EXAMPLE:**

????

### **SCOPE RESOLUTION OPERATOR**

**EXAMPLE:**

```
class Base;  
typedef enum {bin,oct,dec,hex} radix;  
  
task print( radix r, integer n );
```

```

$display(" Enum is %s ",r.name());
$display(" Val is %d",n);
endtask
endclass

program main;
initial
begin
    Base b = new;
    int bin = 123;
    b.print( Base::bin, bin ); // Base::bin and bin are different
end
endprogram
RESULT:

```

Enum is bin

Val is 123

In addition, to disambiguating class scope identifiers, the :: operator also allows access to static members (class properties and methods) from outside the class,

**EXAMPLE:**

```

class Base;
typedef enum {bin,oct,dec,hex} radix;

task print( radix r, integer n );
    $display(" Enum is %s ",r.name());
    $display(" Val is %d",n);
endtask
endclass
program main;
initial
begin
    int bin = 123;
    Base::print( Base::bin, bin ); // Base::bin and bin are different
end
endprogram
RESULT:

```

Enum is bin

Val is 123

Scope resolution operator :: can be used to access to public or protected elements of a superclass from within the derived classes.

**EXAMPLE:**

```
class Base;
    typedef enum {bin,oct,dec,hex} radix;
endclass

class Ext extends Base;
    typedef enum {dec,hex,bin,oct} radix;

task print();
    $display(" Ext classss :: enum values %d %d %d %d ",bin,oct,dec,hex);
    $display(" Base classss :: enum values %d %d %d %d "
",Base::bin,Base::oct,Base::dec,Base::hex);
endtask
endclass

program main;
    initial
    begin
        Ext e;
        e = new();
        e.print();
    end
endprogram
```

**RESULT:**

Ext classss :: enum values	2	3	0	1
Base classss :: enum values	0	1	2	3

In SystemVerilog, the class scope resolution operator applies to all static elements of a class: static class properties, static methods, typedefs, enumerations, structures, unions, and nested class declarations. Class scope resolved expressions can be read (in expressions), written (in assignments or subroutines calls), or triggered off (in event expressions). They can also be used as the name of a type or a method call.

The class scope resolution operator enables the following:

- ⌚ Access to static public members (methods and class properties) from outside the class hierarchy.
- ⌚ Access to public or protected class members of a superclass from within the derived classes.
- ⌚ Access to type declarations and enumeration named constants declared inside the class from outside the class hierarchy or from within derived classes.

**NULL**

The Null is perhaps the most "intelligent" pattern of all. It knows exactly what to do all the time,

every time. It does nothing. The Null is somewhat difficult to describe, since it resides in an abstract hierarchy tree, having no particular place at all, but occupying many roles. It is somewhat like the mathematical concept of zero: it is a placeholder, but is in itself nothing, and has no value. Null Object is a behavioral pattern designed to act as a default value of an object in most OOPs tools. These references need to be checked to ensure they are not null before invoking any methods, because one can't invoke anything on a null reference; this tends to make code less readable. If you forgot to create an object and passed it to method, where the method has some operation on the object, the simulation fails. So, If the method is expecting an object, then check whether the object is created or not else take necessary action. The advantage of this approach over a working default implementation is that a Null Object is very predictable and has no side effects.

#### EXAMPLE

```
class B;  
    task printf();  
        $display(" Hi ");  
    endtask  
endclass  
  
program main;  
    initial  
    begin  
        B b;  
        print(b);  
    end  
endprogram  
  
task print(B b);  
    if(b == null)  
        $display(" b Object is not created ");  
    else  
        b.printf();  
endtask
```

#### RESULT:

b Object is not created

#### EXTERNAL DECLARATION

Class methods and Constraints can be defined in the following places:

- ⌚ inside a class.
- ⌚ outside a class in the same file.
- ⌚ outside a class in a separate file.

The process of declaring an out of block method involves:

- ⌚ declaring the method prototype or constraint within the class declaration with extern qualifier.
- ⌚ declaring the full method or constraint outside the class body.

The extern qualifier indicates that the body of the method (its implementation) or constraint block is to be found outside the declaration.

NOTE : class scope resolution operator :: should be used while defining.

**EXAMPLE:**

```
class B;  
    extern task printf();  
endclass
```

```
task B::printf();  
    $display(" Hi ");  
endtask
```

```
program main;  
    initial  
    begin  
        B b;  
        b = new();  
        b.printf();  
    end  
endprogram
```

**RESULT:**

Hi

## **CLASSES AND STRUCTURES**

class differs from struct in three fundamental ways:

- ⌚ SystemVerilog structs are strictly static objects; they are created either in a static memory location (global or module scope) or on the stack of an automatic task. Conversely, SystemVerilog objects (i.e., class instances) are exclusively dynamic; their declaration does not create the object. Creating an object is done by calling new.
- ⌚ SystemVerilog objects are implemented using handles, thereby providing C-like pointer functionality. But, SystemVerilog disallows casting handles onto other data types; thus, unlike C, SystemVerilog handles are guaranteed to be safe.
- ⌚ SystemVerilog objects form the basis of an Object-Oriented data abstraction that provides true

polymorphism. Class inheritance, abstract classes, and dynamic casting are powerful mechanisms that go way beyond the mere encapsulation mechanism provided by structs.

## **TYPEDEF CLASS**

### **Forward Reference**

A forward declaration is a declaration of a object which the programmer has not yet given a complete definition. The term forward reference is sometimes used as a synonym of forward declaration. However, more often it is taken to refer to the actual use of an entity before any declaration. The SystemVerilog language supports the typedef class construct for forward referencing of a class declaration. This allows for the compiler to read a file from beginning to end without concern for the positioning of the class declaration.

#### **EXAMPLE:**

```
class Y;  
  X x; // refers to Class X, which is not yet defined  
endclass
```

```
class X;  
  int i;  
endclass
```

## **RESULT**

Error : Class X is not defined

When the compiler encounters the handle x of class type X referred to in class Y, it does not yet know the definition for class X since it is later in the file. Thus, compilation fails.

To rectify this situation, typedef is used to forward reference the class declaration.

#### **EXAMPLE:**

```
typedef class X;  
class Y;  
  X x; // refers to Class X, which is not yet defined  
endclass  
class X;  
  int i;  
endclass
```

The typedef of class X allows the compiler to process Y before X is fully defined. Note that typedef cannot be used to forward reference a class definition in another file. This must be done using the inclusion of a header file.

## **Circular Dependency**

A Circular dependency is a situation which can occur in programming languages wherein the definition of an object includes the object itself. One famous example is Linked List.

### **EXAMPLE:**

```
class Y;  
  int i;  
  Y y; // refers to Class Y, which is not yet defined  
endclass
```

As you seen, there is a compilation error. To avoid this situation, typedef is used to forward reference the class declaration and this circular dependency problem can be avoided.

### **PURE**

As we have already seen in the previous topics , a virtual method may or may not be overridden in the derived classes. It means, it is not necessary for a derived class to override a virtual method. But there are times when a base class is not able to define anything meaningful for the virtual method in that case every derived class must provide its own definition of the that method. A pure virtual method is a virtual method that you want to force derived classes to override. If a class has any unoverridden pure virtuals, it is an "abstract class" and you can't create objects of that type.

" pure virtual function " or " pure virtual task " declaration is supposed to represent the fact that the method has no implementation.

There are two major differences between a virtual and a pure virtual function, these are below:

- ⌚ There CAN'T be a definition of the pure virtual function in the base class.
- ⌚ There MUST be a definition of the pure virtual function in the derived class.

### **EXAMPLE:**

```
class Base;  
  pure virtual task disp();  
end class  
  
program main  
initial  
begin  
  Base B;  
  B = new();  
  B.disp();  
end  
endprogram
```

### **RESULT**

Error: pure virtual task disp(); must be overridden in derived class

### **OTHER OOPS FEATURES**

Multiple inheritance and Function overloading and the OOPs features which are not supported by System Verilog.

## Multiple Inheritance

Multiple inheritance refers to a feature of some object-oriented programming languages in which a class can inherit behaviors and features from more than one superclass. This contrasts with single inheritance, where a class may inherit from at most one superclass. SystemC supports multiple inheritance, SystemVerilog supports only single inheritance.

Multiple inheritance allows a class to take on functionality from multiple other classes, such as allowing a class named D to inherit from a class named A, a class named B, and a class named C.

**EXAMPLE:**

```
class A;  
.....  
endclass  
  
class B;  
.....  
endclass  
  
class C;  
.....  
endclass  
  
class D extends A,B,C;  
.....  
endclass
```

Multiple inheritance is not implemented well in SystemVerilog languages . Implementation problems include:

- ⌚ Increased complexity
- ⌚ Not being able to explicitly inherit from multiple times from a single class
- ⌚ Order of inheritance changing class semantics.

## Method Overloading

Method overloading is the practice of declaring the same method with different signatures. The same method name will be used with different data type . This is Not Supported by SystemVerilog as of 2008.

**EXAMPLE:**

```
task my_task(integer i) { ... }  
task my_task(string s) { ... }  
program test {  
    integer number = 10;  
    string name = "vera";
```

```

my_task(number);
my_task(name);
}

```

## MISC

### Always Block In Classes

SystemVerilog does not allow to define always block in program block or class, as these are meant for testbench purpose.

Example to show the implementation of always block in program block.

**EXAMPLE:**

```

program main;

    integer a,b;

    initial
        repeat(4)
            begin
                #({$random()}%20)
                a = $random();
                #({$random()}%20)
                b = $random();
            end
        initial
            always_task();
        task always_task();
        fork
            forever
                begin
                    @(a,b);
                    $display(" a is %d : b is %d at %t ",a,b,$time);
                end
            join_none
        endtask

endprogram

```

## **RESULT**

a is -1064739199 : b is	x at	8
a is -1064739199 : b is -1309649309 at		25
a is 1189058957 : b is -1309649309 at		42

a is 1189058957 : b is -1992863214 at	47
a is 114806029 : b is -1992863214 at	48
a is 114806029 : b is 512609597 at	66
a is 1177417612 : b is 512609597 at	75
a is 1177417612 : b is -482925370 at	84

Example to show the implementation of always block in class.

#### EXAMPLE

```

class Base;
  integer a,b;
    task always_task();
fork
  forever
  begin
    @(a,b);
    $display(" a is %d : b is %d at %t ",a,b,$time);
  end
  join_none
  endtask
  endclass
program main;
  initial
  begin
    Base obj;
    obj = new();
    // start the always block.
    fork
      obj.always_task();
    join_none
    repeat(4)
      begin
        #({$random()%20}
        obj.a = $random();
        #({$random()%20}
        obj.b = $random();
      end
    end
  end
endprogram

```

#### RESULT

a is -1064739199 : b is	x at	8
a is -1064739199 : b is -1309649309	at	25
a is 1189058957 : b is -1309649309	at	42
a is 1189058957 : b is -1992863214	at	47
a is 114806029 : b is -1992863214	at	48
a is 114806029 : b is 512609597	at	66
a is 1177417612 : b is 512609597	at	75
a is 1177417612 : b is -482925370	at	84

## **CONSTRAINED RANDOM VERIFICATION**

### **Introduction :**

Historically, verification engineers used directed test bench to verify the functionality of their design. Rapid changes have occurred during the past decades in design and verification. High Level Verification Languages (HVLs) such as e, System c, Vera, SystemVerilog have become a necessity for verification environments.

Constraint Random stimulus generation is not new. Everybody uses Verilog and VHDL at very low level abstraction for this purpose. HVLs provide constructs to express specification of stimulus at high level of abstraction and constraint solver generates legal stimulus.

Writing constraints at higher level of abstraction, makes the programming closer to spec.

A constraint language should support:

- ⌚ Expressions to complex scenarios.
- ⌚ Flexibility to control dynamically.
- ⌚ Combinational and sequential constraints.

### **EXAMPLE:**

Combinational constraint :

In ethernet, 13 & 14 th bytes should be equal to payload length.

Sequential constraint:

If request comes, then acknowledgement should be given between 4th to 10th cycles.

NOTE: SystemVerilog does not support sequential constraints.

This article is about Constrained random verification using SystemVerilog. I tried to explain every point using examples.

## **VERILOG CRV**

### **Constrained Random Stimulus Generation In Verilog:**

Verilog has system function \$random, which can be used to generate random input vectors.

With this approach, we can generate values which we wouldnt have got, if listed manually. In this topic I would like to discuss what natural things happening behind \$random and how we use it in different manners.

### EXAMPLE:

```
module Tb_mem();
    reg clock;
    reg read_write;
    reg [31:0] data;
    reg [31:0] address;
    initial
        begin
            clock = 0;
            forever #10 clock = ~clock;
        end
    initial
        begin
            repeat(5)@(negedge clock)
                begin
                    read_write = $random; data = $random; address = $random;
                    $display($time, " read_write = %d ; data = %d ; address = %d;", read_write,data,address);
                end
            #10 $finish;
        end
    endmodule
```

### RESULT:

```
20 read_write = 0 ; data = 3230228097 ; address = 2223298057;
40 read_write = 1 ; data = 112818957 ; address = 1189058957;
60 read_write = 1 ; data = 2302104082 ; address = 15983361;
80 read_write = 1 ; data = 992211318 ; address = 512609597;
100 read_write = 1 ; data = 1177417612 ; address = 2097015289;
```

\$random system function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative. The following example demonstrates random generation of signed numbers.

### EXAMPLE:

```
module Tb();
    integer address;

    initial
        begin
            repeat(5)
                #1 address = $random;
        end
```

```
initial
  $monitor("address = %0d;",address);
endmodule
```

### RESULT:

```
address = 303379748;
address = -1064739199;
address = -2071669239;
address = -1309649309;
address = 112818957;
```

We have seen how to generate random numbers. But the numbers range from  $-(2^{32}-1)$  to  $2^{32}$ . Most of the time, the requirement is not to need this range. For example, take a memory. The address starts from 0 to some 1k or 1m. Generating a random address which DUT is not supporting is meaningless. In verilog there are no constructs to constraint randomization. Following example demonstrated how to generate random number between 0 to 10. Using % operation, the remainder of any number is always between 0 to 10.

### EXAMPLE:

```
module Tb();
  integer add_1;
  initial
  begin
    repeat(5)
      begin
        #1;
        add_1 = $random % 10;
      end
    end
    initial
      $monitor("add_1 = %0d",add_1);
  endmodule
```

### RESULT:

```
add_1 = 8;
add_1 = 4294967287;
add_1 = 4294967295;
add_1 = 9;
add_1 = 9;
```

OOPS!..... The results are not what is expected. The reason is \$random generates negative numbers also. The following example demonstrates proper way of generating a random number

between 0 to 10. Concatenation operator returns only bitvector. Bit vectors are unsigned, so the results are correct as we expected. Verilog also has \$unsigned systemtask to convert signed numbers to signed number. This can also be used to meet the requirements. The following example shows the usage of concatenation operator and \$unsigned.

**EXAMPLE:**

```
module Tb();
    integer add_2;
    reg [31:0] add_1;
    integer add_3;

    initial
        begin
            repeat(5)
                begin
                    #1;
                    add_1 = $random % 10;
                    add_2 = {$random} %10 ;
                    add_3 = $unsigned($random) %10 ;
                end
        end

    initial
        $monitor("add_3 = %0d;add_2 = %0d;add_1 = %0d",add_3,add_2,add_1);

endmodule
```

**RESULT:**

```
add_3 = 7;add_2 = 7;add_1 = 8
add_3 = 7;add_2 = 7;add_1 = 4
add_3 = 1;add_2 = 2;add_1 = 4
add_3 = 7;add_2 = 8;add_1 = 9
add_3 = 9;add_2 = 2;add_1 = 9
```

The above example shows the generation of numbers from 0 to N. Some specification require the range to start from non Zero number. MIN + {\$random} % (MAX - MIN ) will generate random numbers between MIN and MAX.

**EXAMPLE:**

```
module Tb();
    integer add;

    initial
        begin
```

```

repeat(5)
  begin
    #1;
    add = 40 + {$random} % (50 - 40) ;
    $display("add = %0d",add);
  end
end
endmodule

```

**RESULT:**

```

add = 48
add = 47
add = 47
add = 47
add = 47

```

Now how to generate a random number between two ranges? The number should be between MIN1 and MAX1 or MIN2 and MAX2. The following example show how to generate this specification.

**EXAMPLE:**

```

module Tb();
  integer add;
  initial
  begin
    repeat(5)
      begin
        #1;
        if($random % 2)
          add = 40 + {$random} % (50 - 40) ;
        else
          add = 90 + {$random} % (100 - 90) ;
        $display("add = %0d",add);
      end
    end
  endmodule

```

**RESULT:**

```

add = 97
add = 47
add = 47
add = 42
add = 49

```

All the random numbers generated above are 32 bit vectors, which is not always the same

requirement. For example, to generate a 5 bit and 45 bit vector random number, the following method can be used.

**EXAMPLE:**

```
module Tb();
    reg [4:0] add_1;
    reg [44:0] add_2;
initial
begin
repeat(5)
begin
    add_1 = $random ;
    add_2 = {$random,$random};
    $display("add_1 = %b,add_2 = %b ",add_1,add_2);
end
end
endmodule
```

**RESULTS:**

add\_1 = 00100,add\_2 = 111101000000110000100100001001101011000001001

add\_1 = 00011,add\_2 = 11011000011010100011011011111001100110001101

add\_1 = 00101,add\_2 = 100100001001000000001110011110001100000001

add\_1 = 01101,add\_2 = 1000101101100001110100011011100110100111101

add\_1 = 01101,add\_2 = 10111000110001111001111101111010011111001

Some protocols require a random number which is multiple to some number. For example, Ethernet packet is always in multiples of 8bits. Look at the following example. It generates a random number which is multiple of 3 and 5.

\$random \* 3 will give random numbers which are multiples of 3. But if the number after multiplication needs more than 32 bit to represent, then the results may be wrong.

**EXAMPLE:**

```
module Tb();
integer num_1,num_2,tmp;
initial
begin
repeat(5)
begin
    num_1 =($random / 3)*3;
    num_2 =($random / 5)*5;
    $display("num_1 = %d,num_2 = %d",num_1,num_2);
end
end
```

**endmodule**

**RESULT:**

```
num_1 = 303379746,num_2 = -1064739195  
num_1 = -2071669239,num_2 = -1309649305  
num_1 = 112818957,num_2 = 1189058955  
num_1 = -1295874969,num_2 = -1992863210  
num_1 = 15983361,num_2 = 114806025
```

All the above examples show that the random numbers are integers only. In verilog there is no special construct to generate a random real number. The following method shows generation of random real numbers.

**EXAMPLE:**

```
module Tb();  
integer num_1,num_2,num_3;  
real r_num;  
initial  
begin  
repeat(5)  
begin  
#1;  
num_1 = $random;  
num_2 = $random;  
num_3 = $random;  
r_num = num_1 + ((10)**(-(num_2)))*(num_3);  
$display("r_num = %e",r_num);  
end  
end  
endmodule
```

**RESULT:**

```
r_num = -2.071669e+03  
r_num = 2641.189059e+013  
r_num = 976361.598336e+01  
r_num = 57645.126096e+02  
r_num = 24589.097015e+0
```

To generate random real number , system function \$bitstoreal can also be used.

**EXAMPLE:**

```
module Tb();
```

```

real r_num;
initial
begin
    repeat(5)
        begin
            #1;
            r_num = $bitstoreal({$random,$random});
            $display("r_num = %e",r_num);
        end
    end
endmodule

```

#### RESULTS:

```

r_num = 1.466745e-221
r_num = -6.841798e-287
r_num = 2.874848e-276
r_num = -3.516622e-64
r_num = 4.531144e-304

```

If you want more control over randomizing real numbers in terms of sign, exponential and mantissa, use \$bitstoreal() as shown in example below. For positive numbers use sgn = 0 and for negative numbers use sgn = 1 .

#### EXAMPLE:

```

module Tb();
    reg sgn;
    reg [10:0] exp;
    reg [51:0] man;
    real r_num;

    initial
    begin
        repeat(5)
            begin
                sgn = $random;
                exp = $random;
                man = $random;
                r_num = $bitstoreal({sgn,exp,man});
                $display("r_num = %e",r_num);
            end
    end
endmodule

```

## RESULTS:

```
r_num = 3.649952e+193
r_num = -1.414950e-73
r_num = -3.910319e-149
r_num = -4.280878e-196
r_num = -4.327791e+273
```

Some times it is required to generate random numbers without repetition. The random numbers should be unique. For example,to generate 10 random numbers between 0 to 9 without repetition, the following logic can be used.

## EXAMPLE:

```
module Tb();
    integer num,i,j,index;
    integer arr[9:0];
    reg ind[9:0];
    reg got;
    initial
        begin
            index=0;
            for(i=0;i<10;i=i+1)
                begin
                    arr[i] = i;
                    ind[i] = 1;
                end
            for(j = 0;j<10 ;j=j+1)
                begin
                    got = 0;
                    while(got == 0)
                        begin
                            index = { $random } % 10;
                            if(ind[index] == 1)
                                begin
                                    ind[index] = 0;
                                    got = 1;
                                    num = arr[index];
                                end
                            end
                        end
                    $write("| num=%2d |",num);
                end
            end
endmodule
```

## **RESULT:**

```
| num= 8 || num= 7 || num= 5 || num= 2 || num= 1 || num= 9 || num= 6 || num= 4 || num= 0 || num= 3 |
```

Random number returned by \$random system function should be deterministic, i.e when ever we run with simulator it should return values in same sequence. Otherwise the bug found today cant be found return. For this purpose it has one argument called seed. The seed parameter controls the numbers that \$random returns, such that different seeds generate different random streams. The seed parameter shall be either a reg, an integer, or a time variable. The seed value should be assigned to this variable prior to calling \$random.

## **EXAMPLE:**

```
module Tb();
integer num,seed,i,j;
initial
begin
for(j = 0;j<4 ;j=j+1)
begin
    seed = j;
    $display(" seed is %d",seed);
    for(i = 0;i < 10; i=i+1)
    begin
        num = { $random(seed) } % 10;
        $write("| num=%2d |",num);
    end
    $display(" ");
end
end
endmodule
```

## **RESULT:**

seed is 0

```
| num= 8 || num= 7 || num= 7 || num= 7 || num= 7 || num= 5 || num= 2 || num= 1 || num= 9 |
```

seed is 1

```
| num= 8 || num= 8 || num= 2 || num= 2 || num= 6 || num= 3 || num= 8 || num= 5 || num= 5 |
```

seed is 2

```
| num= 8 || num= 1 || num= 0 || num= 5 || num= 0 || num= 8 || num= 6 || num= 7 || num= 1 || num= 6 |
```

seed is 3

```
| num= 8 || num= 2 || num= 2 || num= 3 || num= 8 || num= 6 || num= 1 || num= 4 || num= 3 || num= 9 |
```

The \$random function has its own implicit variable as seed when user is not giving explicitly seed. The following example shows that seed = 0 and implicit seed are having same sequence. It means that the imlicity taken seed is also 0.

**EXAMPLE:**

```
module Tb();
integer num,seed,i,j;
initial
begin
    seed = 0;
    for(j = 0;j<2 ;j=j+1)
begin
    if(j ==0)
        $display(" seed is %d",seed);
    else
        $display(" No seed is given ");
    for(i = 0;i < 10; i=i+1)
begin
    if( j == 0)
        num = { $random(seed) } % 10;
    else
        num = { $random } % 10;
    $write("| num=%2d |",num);
end
    $display(" ");
end
end
endmodule
```

**RESULT:**

```
seed is 0
| num= 8 || num= 7 || num= 5 || num= 2 || num= 1 || num=
9 |
No seed is given
| num= 8 || num= 7 || num= 5 || num= 2 || num= 1 || num=
9 |
```

The system functions shall always return same series of values with same seed. This facilitates debugging by making the operation of the system repeatable. The argument for the seed parameter should be an integer variable that is initialized by the user and only updated by the system function. This ensures the desired distribution is achieved. In the following example, when ever the seed is changed to 2, the sequence 8-1-0-5-0-8-6-7-11-6..... is followed. Check out in any tool you will see the same sequence.

**EXAMPLE:**

```
module Tb();
integer num,seed,i,j;
initial
begin
  for(j = 0;j<4 ;j=j+1)
begin
  seed = 2;
  $display(" seed is %d",seed);
  for(i = 0;i < 10; i=i+1)
  begin
    num = { $random(seed) } % 10;
    $write("| num=%2d |",num);
  end
  $display(" ");
end
end
endmodule
```

**RESULT:**

```
seed is 2
| num= 8 || num= 1 || num= 0 || num= 5 || num= 0 || num= 8 || num= 6 || num= 7 || num= 1 || num=
6 |
seed is 2
| num= 8 || num= 1 || num= 0 || num= 5 || num= 0 || num= 8 || num= 6 || num= 7 || num= 1 || num=
6 |
seed is 2
| num= 8 || num= 1 || num= 0 || num= 5 || num= 0 || num= 8 || num= 6 || num= 7 || num= 1 || num=
6 |
seed is 2
| num= 8 || num= 1 || num= 0 || num= 5 || num= 0 || num= 8 || num= 6 || num= 7 || num= 1 || num=
6 |
```

Seed is inout port. Random number system function returns a random number and also returns a random number to seed inout argument also. The results of the following example demonstrates how seed value is getting changed.

**EXAMPLE:**

```
module Tb();
integer num,seed,i,j;
initial
begin
  seed = 0;
```

```

for(j = 0;j<10 ;j=j+1)
begin
    num = { $random(seed) } % 10;
    $write("| num=%2d |",num);
    $display(" seed is %d ",seed);
end
end
endmodule

```

**RESULT:**

```

| num= 8 | seed is -1844104698
| num= 7 | seed is 1082744015
| num= 7 | seed is 75814084
| num= 7 | seed is 837833973
| num= 7 | seed is -2034665166
| num= 7 | seed is -958425333
| num= 5 | seed is 851608272
| num= 2 | seed is 154620049
| num= 1 | seed is -2131500770
| num= 9 | seed is -2032678137

```

From the above results we can make a table of seed values and return values of \$random. If a seed is taken from table, then rest of the sequence has to follow sequence in table.

Table is as follows for initial seed 0;

```

| num= 8 | seed is -1844104698
| num= 7 | seed is 1082744015
| num= 7 | seed is 75814084
| num= 7 | seed is 837833973
| num= 7 | seed is -2034665166
| num= 7 | seed is -958425333
| num= 5 | seed is 851608272
| num= 2 | seed is 154620049
| num= 1 | seed is -2131500770
| num= 9 | seed is -2032678137

```

.

.

.

table goes on.....

In the following example, the seed is 837833973, which is the 4 th seed from the above table.

### EXAMPLE:

```
module Tb();
integer num,seed,i,j;
initial
begin
    seed = 837833973;
    for(j = 0;j<10 ;j=j+1)
begin
    num = { $random(seed) } % 10;
    $write("| num=%2d |",num);
    $display(" seed is %d ",seed);
end
end
endmodule
```

### RESULTS:

```
| num= 7 | seed is -2034665166
| num= 7 | seed is -958425333
| num= 5 | seed is 851608272
| num= 2 | seed is 154620049
| num= 1 | seed is -2131500770
| num= 9 | seed is -2032678137
| num= 8 | seed is -1155272804
| num= 7 | seed is -1634874387
| num= 9 | seed is -153856566
| num= 2 | seed is -970066749
```

From the above example we can come to conclusion that \$random is not giving a random number. It is randomizing seed and returning corresponding number for that seed.

Total possible seed values are 4294967295(32'hFFFF\_FFFF). Is it possible for \$random to generate all the seeds? . Lets say, if the seed gets repeated after 10 iterations, then after the 10 iterations, same values are repeated. So \$random is circulating inside a chain of 10 numbers.

The following example demonstrates how \$random misses many seeds. I tried to display the seeds between 0 to 20 in the chaining formed by initial seed of 0. Results show that total possible seeds are 4294967295 , and number of seeds possible in seed chain are 4030768279 , so we are missing some seeds. Look at the seeds between 0 to 20. Seed == 1 is missing.

### EXAMPLE:

```
module Tb();
integer num,seed,j;
reg [0:31] i;
initial
```

```

begin
    i = 0;
    seed = 1;
    while (seed != 0)
        begin
            if(i == 0)
                seed = 0;
                i = i + 1;
                num = $random(seed);
            if(seed < 20 && seed > 0)
                $display(" seed is %d after values %d ",seed,i);
        end
        $display(" seed is zero after this number of random numbers %0d total numbers available are
%d",i,{32'hffff_ffff});
    end
endmodule

```

#### **RESULTS:**

```

seed is 10 after values 93137101
seed is 17 after values 307298440
seed is 2 after values 410139893
seed is 12 after values 483530075
seed is 19 after values 592243262
seed is 3 after values 720224974
seed is 11 after values 1342230278
seed is 15 after values 2032553666
seed is 7 after values 2266624778
seed is 13 after values 2362534380
seed is 5 after values 2512466932
seed is 9 after values 2575033104
seed is 16 after values 2988686279
seed is 4 after values 3173376451
seed is 6 after values 3483433473
seed is 8 after values 3547878575
seed is 14 after values 3663208793
seed is 18 after values 3930700709
seed is zero after this number of random numbers 4030768279 total numbers available are
4294967295

```

Now I tried to simulate with seed = 1 . It's interesting to know that some how the sequence is able to enter this chaining which is formed with seed=0 and there is no seed value 1 in this chaining and my simulation hanged. So aborted the simulation and partial results show that the initial seed = 1 enters the chain formed by seed 0.

**EXAMPLE:**

```
module Tb();
integer num,seed,j;
reg [0:31] i;
initial
begin
    i = 0;
    seed = 0;
    while (seed != 1)
begin
    if(i == 0)
        seed = 1;
    i = i + 1;
    num = $random(seed);
    if(seed < 20 && seed > 0)
        $display(" seed is %d after values %d ",seed,i);
end
$display(" seed is one after this number of random numbers %0d total numbers available are
%d",i,{32'hffff_ffff});
end
endmodule
```

**RESULTS:**

```
seed is 10 after values 357336117
seed is 17 after values 571497456
seed is 2 after values 674338909
seed is 12 after values 747729091
seed is 19 after values 856442278
seed is 3 after values 984423990
seed is 11 after values 1606429294
seed is 15 after values 2296752682
seed is 7 after values 2530823794
seed is 13 after values 2626733396
seed is 5 after values 2776665948
seed is 9 after values 2839232120
seed is 16 after values 3252885295
seed is 4 after values 3437575467
```

```
seed is 6 after values 3747632489
seed is 8 after values 3812077591
seed is 14 after values 3927407809
seed is 18 after values 4194899725
seed is 10 after values 357336117
seed is 17 after values 571497456
seed is 2 after values 674338909
seed is 12 after values 747729091
seed is 19 after values 856442278
seed is 3 after values 984423990
```

Verilog also has other system functions to generate random numbers. Each of these functions returns a pseudo-random number whose characteristics are described by the function name.

Following are the Verilog random number generator system functions:

```
$random
$dist_chi_square
$dist_erlang
$dist_exponential
$dist_normal
$dist_poisson
$dist_t
$dist_uniform
```

All parameters to the system functions are integer values. For the exponential, poisson, chi-square ,t and erlang functions the parameters mean, degree of freedom, and k\_stage must be greater than 0.

\$dist\_uniform(seed, min, max) is similar to  $\min + \{\$random(\text{seed})\} \% (\max - \min + 1)$ , the difference is that in \$dist\_uniform the distribution is uniform. \$dist\_uniform returns a number between min and max. In the \$dist\_uniform function, the start and end parameters are integer inputs that bound the values returned. The start value should be smaller than the end value.

The mean parameter used by \$dist\_normal, \$dist\_exponential, \$dist\_poisson and \$dist\_erlang is an integer input that causes the average value returned by the function to approach the value specified. The standard deviation parameter used with the \$dist\_normal function is an integer input that helps determine the shape of the density function. Larger numbers for standard deviation spread the returned values over a wider range.

The degree of freedom parameter used with the \$dist\_chi\_square and \$dist\_t functions is an integer input that helps determine the shape of the density function. Larger numbers spread the returned values over a wider range.

#### EXAMPLE:

```
module Tb();
integer num_1,num_2,seed;
initial
```

```

begin
  seed = 10;
  repeat(5)
    begin
      #1;
      num_1 = $dist_uniform(seed,20,25);
      num_2 = $dist_uniform(seed,50,55);
      $display("num_1 = %d,num_2 = %d",num_1,num_2);
    end
  end
endmodule

```

#### **RESULTS:**

```

num_1 = 20,num_2 = 50
num_1 = 23,num_2 = 55
num_1 = 22,num_2 = 54
num_1 = 25,num_2 = 51
num_1 = 23,num_2 = 55

```

As i discussed \$random randomizes its seed, Lets see whether \$dist\_uniform is also doing the same.

#### **EXAMPLE:**

```

module Tb();
  integer num_1,num_2,seedd,seedr;
  initial
  begin
    seedd = 10;
    seedr = 10;
    repeat(5)
      begin
        #1;
        num_1 = $dist_uniform(seedd,20,25);
        num_2 = 20 + ({$random(seedr)} % 6);
        $display("num_1 = %d,num_2 = %d,seedd = %d seedr = %d",num_1,num_2,seedd,seedr);
      end
    end
  endmodule

```

#### **RESULTS:**

```

num_1 = 20,num_2 = 22,seedd = 690691 seedr = 690691
num_1 = 20,num_2 = 20,seedd = 460696424 seedr = 460696424
num_1 = 23,num_2 = 22,seedd = -1571386807 seedr = -1571386807
num_1 = 25,num_2 = 21,seedd = -291802762 seedr = -291802762

```

num\_1 = 22,num\_2 = 23,seedd = 1756551551 seedr = 1756551551

Look at the results... Its interesting to note that \$random and \$dist\_uniform have same seed sequence flow also.

As I mentioned, \$dist\_uniform(seed, min, max) is similar to min + {\$random(seed)}%(max-min+1). "similar" means they have some common functionality. \$dist\_uniform is having uniform distribution, \$random for that range is also uniformly distributed. Following example demonstrates that \$dist\_uniform and \$random are uniformly distributed.

**EXAMPLE:**

```
module Tb();
integer num,seed;
integer num_20,num_21,num_22,num_23,num_24,num_25;
initial
begin
seed = 10;
num_20 = 0;num_21 = 0;num_22 = 0;num_23 = 0;num_24 = 0;num_25 = 0;
repeat(6000)
begin
num = $dist_uniform(seed,20,25);
if(num == 20 )
    num_20 = num_20 + 1;
if(num == 21)
    num_21 = num_21 + 1;
if(num == 22)
    num_22 = num_22 + 1;
if(num == 23)
    num_23 = num_23 + 1;
if(num == 24)
    num_24 = num_24 + 1;
if(num == 25)
    num_25 = num_25 + 1;
end
$display("num_20 = %0d;num_21 = %0d;num_22 = %0d;num_23 = %0d;num_24 =
%0d;num_25 = %0d",num_20,num_21,num_22,num_23,num_24,num_25);
end
endmodule
```

**RESULTS:**

num\_20 = 1014;num\_21 = 983;num\_22 = 946;num\_23 = 1023;num\_24 = 1014;num\_25 = 1020

**EXAMPLE:**

```
module Tb();
integer num;
```

```

integer num_20,num_21,num_22,num_23,num_24,num_25;
initial
begin
    seed = 10;
    num_20 = 0;num_21 = 0;num_22 = 0;num_23 = 0;num_24 = 0;num_25 = 0;
    repeat(6000)
begin
    num = 20 + ( {$random(seed) } %6 );
    if(num == 20 )
        num_20 = num_20 + 1;
    if(num == 21)
        num_21 = num_21 + 1;
    if(num == 22)
        num_22 = num_22 + 1;
    if(num == 23)
        num_23 = num_23 + 1;
    if(num == 24)
        num_24 = num_24 + 1;
    if(num == 25)
        num_25 = num_25 + 1;
end
$display("num_20 = %0d;num_21 = %0d;num_22 = %0d;num_23 = %0d;num_24 =
%0d;num_25 = %0d",num_20,num_21,num_22,num_23,num_24,num_25);
end
endmodule

```

### RESULTS:

num\_20 = 973;num\_21 = 1064;num\_22 = 961;num\_23 = 988;num\_24 = 999;num\_25 = 1015

As I mentioned ,\$dist\_uniform(seed, min, max) is similar to min + {\$random(seed)}%(max-min+1). "similar" means they have some difference. The difference is that they generate different sequence.

### EXAMPLE:

```

module Tb();
integer num_1,num_2,seedd,seedr;
initial
begin
    seedd = 10;
    seedr = 10;
    repeat(5)
begin
    #1;

```

```

num_1 = $dist_uniform(seedd,20,25);
num_2 = 20 + ({$random(seedr)} % 6);
$display("num_1 = %d,num_2 = %d",num_1,num_2);
end
end
endmodule

```

### RESULTS:

```

num_1 = 20,num_2 = 22
num_1 = 20,num_2 = 20
num_1 = 23,num_2 = 22
num_1 = 25,num_2 = 21
num_1 = 22,num_2 = 23

```

Till now what we have seen is \$random has uniform distribution over integer values. It means that distribution should be uniform across all the bits in 32 bit vector also. The following example shows that bits positions 2,3,4,11,12,13 have equal probability of getting 0. For demonstration I showed some index only. Try out rest of them and see that results is same for all the bits.

### EXAMPLE:

```

module Tb();
integer num;
integer num_2,num_3,num_4,num_11,num_12,num_13;
initial
begin
    num_2 = 0;num_3 = 0;num_4 = 0;num_11 = 0;num_12 = 0;num_13 = 0;
    repeat(6000)
        begin
            num = $random;
            if(num[2] == 0 )
                num_2 = num_2 + 1;
            if(num[3] == 0)
                num_3 = num_3 + 1;
            if(num[4] == 0)
                num_4 = num_4 + 1;
            if(num[11] == 0)
                num_11 = num_11 + 1;
            if(num[12] == 0)
                num_12 = num_12 + 1;
            if(num[13] == 0)
                num_13 = num_13 + 1;
        end

```

```

$display("num_2 = %0d;num_3 = %0d;num_4 = %0d;num_11 = %0d;num_12 =
%0d;num_13 = %0d",num_2,num_3,num_4,num_11,num_12,num_13);
end
endmodule

```

### **RESULTS:**

num\_2 = 3012;num\_3 = 2964;num\_4 = 3065;num\_11 = 3001;num\_12 = 2964;num\_13 = 2975

The distribution is uniform for system function \$random. Suppose if the requirement is to generate random numbers for more than one variable, and all the variables should have uniform distribution, then use different seeds for each variable. Other wise distribution is distributed on all the variables as overall variables which mightnot be our requirement.. But for lower bits, the distribution is same as shown in example.

### **EXAMPLE:**

```

module Tb();
integer seed;
reg [1:0] var_1,var_2,var3,var4;
integer num_2,num_3,num_1,num_0;
integer cou_2,cou_3,cou_1,cou_0;

initial
begin
    seed = 10;
    num_2 = 0;num_3= 0;num_1= 0;num_0= 0;
    cou_2= 0;cou_3= 0;cou_1= 0;cou_0= 0;
    repeat(40000)
        begin
            var_1 = $random;
            var3 = $random;
            var4 = $random;
            var_2 = $random;
            if(var_1 == 0 )
                num_0 = num_0 + 1;
            if(var_1 == 1 )
                num_1 = num_1 + 1;
            if(var_1 == 2 )
                num_2 = num_2 + 1;
            if(var_1 == 3 )
                num_3 = num_3 + 1;

            if(var_2 == 0 )

```

```

cou_0 = cou_0 + 1;
if(var_2 == 1 )
    cou_1 = cou_1 + 1;
if(var_2 == 2 )
    cou_2 = cou_2 + 1;
if(var_2 == 3 )
    cou_3 = cou_3 + 1;
end
$display("num_2 = %0d;num_3= %0d;num_1= %0d;num_0=
%0d;",num_2,num_3,num_1,num_0);
$display("cou_2= %0d;cou_3= %0d;cou_1= %0d;cou_0=
%0d;",cou_2,cou_3,cou_1,cou_0);
end
endmodule

```

### RESULTS:

num\_2 = 9984;num\_3= 10059;num\_1= 10002;num\_0= 9955;  
cou\_2= 10060;cou\_3= 9934;cou\_1= 10072;cou\_0= 9934;

Use system time as seed, so the same TB simulated at different times have different random sequences and there is more probability of finding bugs. The following is C code useful in PLI to get system time in to verilog.

```

#include <stdio.h>
#include <time.h>
char *get_time_string(int mode24);
int get_systime() {
time_t seconds;
seconds = time (NULL);
return seconds;
}

```

In Verilog 1995 standard every simulator has its own random number generation algorithm. But in Verilog 2001 a standard is made that every simulator has to follow same algorithm. So the same random number **sequence** can seen on different simulators **for** same seed.

Don't **expect** that the same **sequence** is generated on all the simulators. They are only following same algorithm. The reason is race condition. Look at following example, both the statements\_1 **and** statement\_2 are scheduled to execute at same simulation **time**. The order of execution is **not not** known. Some simulators take statement\_1 as the **first** statement to execute **and** some other statement\_2. If the **TB** is built without any race condition to **\$random function** calls, then the same random **sequence** can be generated on different simulators **and** a testbench without a racecondition on **\$random** calls is **not** easy to build.

Look at the following 2 examples. I just changed the order of statements, the results are reversed.

@edes

**EXAMPLE:new**

```
module Tb();
    initial
        $display("staement 2 :::%d",$random);
    initial
        $display("staement 1 :::%d",$random);
endmodule
```

staement 2 ::: 303379748

staement 1 :::-1064739199

**EXAMPLE:**

```
module Tb();
    initial
        $display("staement 1 :::%d",$random);
    initial
        $display("staement 2 :::%d",$random);
endmodule
```

staement 1 ::: 303379748

staement 2 :::-1064739199

## **SYSTEMVERILOG CRV**

### **Systemverilog Constraint Random Stmulus Generaion :**

We have seen how to get random values and constrain them. These constraints are at very low level of abstraction. Todays verification needs a better way to describ the constraints.

SystemVerilog has randomization constructs to support todays verification needs.

Following are the features of SystemVerilog which support Constraint Random Verification (CRV) :

① Constraints : Purely random stimulus takes too long to generate interesting senarios. Specify the interesting subset of all possible stimulus with constraint blocks. These are features provided by SystemVerilog for constraining randomness. Random variable generated in verilog Boolean expressions, foreach (for constraining elements of array), set membership, inline constraints, rand case, rand sequence, Conditional constraints and implication constraints.

② Randomization : random function, constrained and unconstrained randomization, uniform distribution, weighted distribution,weighted range, weighted case, pre randomization, post randomization, declaration of random variable and non repeating sequence.

③ Dynamic constraints : inline constraints, guarded constraints, disabling/enabling constraints, disabling/enabling random variables and overriding of constraint blocks.

☞ 4) Random Stability : Thread stability, object stability and manual seeding.

In verilog only system functions like \$random are used for stimulus generation.

In SystemVerilog, constraint random stimulus can be generated in following ways.

### **Random Number Generator System Functions**

In addition to the system function which are in verilog, SystemVerilog has \$urandom() and \$urandom\_range(). \$urandom() and \$urandom\_range() returns a unsigned values.

The following example demonstrates random generation of unsigned numbers.

**EXAMPLE:**

```
module Tb();
    integer unsigned address;
    initial
    begin
        repeat(5)
        begin
            address = $urandom();
            $display("address = %d;",address);
        end
    end
    endmodule
```

**RESULTS:**

```
# address = 3468265955;
# address = 541037099;
# address = 3836988758;
# address = 3611785217;
# address = 978699393;
```

The seed is an optional argument that determines the sequence of random numbers generated.

The seed can be any integral expression. The random number generator (RNG) shall generate the same sequence of random numbers every time the same seed is used.

**EXAMPLE:**

```
module Tb();
    integer num,seed,i,j;
    initial
    begin
        for(j = 0;j<4 ;j=j+1)
        begin
            seed = 2;
            $display(" seed is set %d",seed);
            void'($urandom(seed));
        for(i = 0;i < 10; i=i+1)
```

```

begin
    num = $urandom() % 10;
    $write("|\t num=%2d |",num);
end
$display(" ");
end
end
endmodule

RESULTS:
seed is set 2
| num= 1 || num= 2 || num= 7 || num= 2 || num= 1 || num= 7 || num= 4 || num= 2 || num= 3 || num=
1 |
seed is set 2
| num= 1 || num= 2 || num= 7 || num= 2 || num= 1 || num= 7 || num= 4 || num= 2 || num= 3 || num=
1 |
seed is set 2
| num= 1 || num= 2 || num= 7 || num= 2 || num= 1 || num= 7 || num= 4 || num= 2 || num= 3 || num=
1 |
seed is set 2
| num= 1 || num= 2 || num= 7 || num= 2 || num= 1 || num= 7 || num= 4 || num= 2 || num= 3 || num=
1 |

$Urandom_range
The $urandom_range() function returns an unsigned integer within a specified range.
The syntax for $urandom_range() is as follows:
function int unsigned $urandom_range( int unsigned maxval,int unsigned minval = 0 );
The function shall return an unsigned integer in the range of maxval ... minval.

EXAMPLE:
module Tb();
    integer num_1,num_2;
    initial
    begin
        repeat(5)
            begin
                #1;
                num_1 = $urandom_range(25,20);
                num_2 = $urandom_range(55,50);
                $display("num_1 = %0d,num_2 = %0d",num_1,num_2);
            end
        end
    endmodule

```

## RESULTS:

```
# num_1 = 25,num_2 = 55
# num_1 = 22,num_2 = 55
# num_1 = 23,num_2 = 52
# num_1 = 21,num_2 = 54
# num_1 = 25,num_2 = 54
```

If minval is omitted, the function shall return a value in the range of maxval ... 0.

## EXAMPLE:

```
module Tb();
integer num_1,num_2;
initial
begin
repeat(5)
begin
#1;
num_1 = $urandom_range(3);
num_2 = $urandom_range(5);
$display("num_1 = %0d,num_2 = %0d",num_1,num_2);
end
end
endmodule
```

## RESULTS:

```
num_1 = 3,num_2 = 5
num_1 = 2,num_2 = 5
num_1 = 1,num_2 = 2
num_1 = 3,num_2 = 4
num_1 = 1,num_2 = 4
```

If maxval is less than minval, the arguments are automatically reversed so that the first argument is larger than the second argument.

## EXAMPLE:

```
module Tb();
integer num_1,num_2;
initial
begin
repeat(5)
begin
#1;
num_1 = $urandom_range(20,25);
num_2 = $urandom_range(50,55);
$display("num_1 = %0d,num_2 = %0d",num_1,num_2);
end
end
endmodule
```

```
    end
  end
endmodule
```

#### RESULTS:

```
num_1 = 25,num_2 = 55
num_1 = 22,num_2 = 55
num_1 = 23,num_2 = 52
num_1 = 21,num_2 = 54
num_1 = 25,num_2 = 54
```

#### Scope Randomize Function

The scope randomize function, randomize(), enables users to randomize data in the current scope. Variables which are passed as arguments are randomized and there is no limit on the number of arguments. For simpler applications, randomize() function leads to straight forward implementation. This gives better control over the \$random, as it allows to add constraints using inline constraints and constraint solver gives valid solution. Variables which are in the constraint block and not passed as arguments to randomize() function are not randomized. In the following example Variable Var is randomized and MIN is not randomized.

#### EXAMPLE:

```
module scope_3;
  integer Var;
  initial
  begin
    for ( int i = 0;i<6 ;i++)
      if( randomize(Var))
        $display(" Randomization sucessfull : Var = %0d ",Var);
      else
        $display("Randomization failed");
    $finish;
  end
endmodule
```

#### RESULTS:

```
Randomization sucessfull : Var = -826701341
Randomization sucessfull : Var = 541037099
Randomization sucessfull : Var = -457978538
Randomization sucessfull : Var = -683182079
Randomization sucessfull : Var = 978699393
Randomization sucessfull : Var = 717199556
Randomization sucessfull : Var = 1114265683
```

Scope randomize function gives better control over the \$random, as it allows to add constraints using inline constraints and constraint solver gives valid solution. Variables which are in the constraint block and not passed as arguments to randomize() function are not randomized. In the following example Variable Var is randomized and MIN is not randomized.

**EXAMPLE:**

```
module scope_4;
    integer Var;
    integer MIN;
    initial
    begin
        MIN = 50;
        for ( int i = 0;i<100 ;i++)
            if( randomize(Var) with { Var < 100 ; Var > MIN ;})
                $display(" Randomization sucessfull : Var = %0d Min = %0d",Var,MIN);
            else
                $display("Randomization failed");
        $finish;
    end
endmodule
```

**RESULTS:**

```
# Randomization sucessfull : Var = 94 Min = 50
# Randomization sucessfull : Var = 69 Min = 50
# Randomization sucessfull : Var = 53 Min = 50
# Randomization sucessfull : Var = 71 Min = 50
# Randomization sucessfull : Var = 51 Min = 50
# Randomization sucessfull : Var = 78 Min = 50
# Randomization sucessfull : Var = 95 Min = 50
```

In randomize function, the solver can't solve if X or Z is used. randomize(Var) with { Var == 'bx' ;} or {MIN = 'bx'} will result in an error.

### **Randomizing Objects**

Generating random stimulus within objects :

SystemVerilog allows object-oriented programming for random stimulus generation, subjected to specified constraints. During randomization, variables declared as rand or randc inside class are only considered for randomization. Built-in randomize() method is called to generate new random values for the declared random variables.

**EXAMPLE:**

```
program Simple_pro_5;
    class Simple;
        rand integer Var;
```

```

endclass
Simple obj;
initial
begin
    obj = new();
    repeat(5)
        if(obj.randomize())
            $display(" Randomization successful : Var = %0d ",obj.Var);
        else
            $display("Randomization failed");
    end
endprogram

```

#### RESULTS:

```

# Randomization sucessfull : Var = -82993358
# Randomization sucessfull : Var = -112342886
# Randomization sucessfull : Var = -867551972
# Randomization sucessfull : Var = -34537722
# Randomization sucessfull : Var = 1977312553

```

#### Random Unpacked Structs:

SystemVerilog allows unpackedstructs to be declared as rand for randomization. Only members of struct which are declared as rand or randc are only randomized. randc is not allowed on unpacked structs. If Struct is not declared as rand, solver considers it as state variable.

#### EXAMPLE:

```

class Rand_struct;
typedef struct {
    randc int Var1;
    int Var2;
} Struct_t;
rand Struct_t Str;      // To randomize Var1 and Struct_t type has to declared as rand
endclass

program stru_rand_6;
Rand_struct RS ;
initial
begin
    RS = new();
    repeat(10)
        if(RS.randomize())
            $display(" Var1 : %d",RS.Str.Var1);
    end
endprogram

```

## RESULTS:

```
# Var1 : -430761355
# Var1 : 424439941
# Var1 : -1129955555
# Var1 : 1781908941
# Var1 : -752252755
# Var1 : 922477361
# Var1 : -2115981855
# Var1 : 1551031961
# Var1 : -91393015
# Var1 : 262093271
```

Simillar to struct, the same can be achived using class by calling the randomize() function on the object, which is created by using class.

## EXAMPLE:

```
program class_rand_7;
  class Class_t;
    rand int Var1;
    int Var2;
  endclass

  class Rand_class;
    rand Class_t Cla; // To randomize Var1,Class_t type has to declared as rand
    function new();
      Cla = new();
    endfunction
  endclass
  Rand_class RC = new();
initial
  repeat(10)
    if(RC.randomize())
      $display(" Var1 : %0d Var2 : %0d",RC.Cla.Var1,RC.Cla.Var2);
endprogram
```

## RESULTS:

```
# Var1 : 733126180 Var2 : 0
# Var1 : -119008195 Var2 : 0
# Var1 : 342785185 Var2 : 0
# Var1 : 679818185 Var2 : 0
# Var1 : -717162992 Var2 : 0
# Var1 : 664520634 Var2 : 0
# Var1 : -1944980214 Var2 : 0
```

```
# Var1 : -1350759145 Var2 : 0
# Var1 : -1374963034 Var2 : 0
# Var1 : -462078992 Var2 : 0
```

SystemVerilog structs are static objects, whereas class instances are dynamic objects, declaring a class instance does not allocate memory space for object. Calling built in new() function creates memory for object. Class have built in functions and tasks, whereas struct don't, this speeds up simulation time if structs are used. Check your data structure, if they need simple encapsulation use struct otherwise if they need object oriented mechanisms then choose class.

### **Rand Case :**

You can use randcase to make a weighted choice between different items, without having to create a class and instance. An item's weight divided by the sum of all weights gives the probability of taking that branch. More details are discussed in the following units.

### **Rand Sequence :**

SystemVerilog provides randsequence to generate random sequences of operation. It will be useful for randomly generating structured sequences of stimulus such as instructions or network traffic patterns.

## **RANDOMIZING OBJECTS**

### **Generating Random Stimulus Within Class :**

SystemVerilog features which support constraint random generation inside objects are :

- ⌚ 1) Random Variable declaration.
- ⌚ 2) Built in Functions for generation.
- ⌚ 3) Constraints to control random generation.

Variables declared as rand or randc are assigned random values when randomize() function is called, where the constraint specifies the valid solution space from which the random values are picked.

## **RANDOM VARIABLES**

### **Random Variable Declaration:**

Variables declared as rand or randc are only randomized due to call of randomize() function. All other variables are considered as state variables.

### **EXAMPLE:**

```
class ex_8;
    rand [3:0] var1;
    randc [3:0] var2;
    rand integer var3;
endclass
```

Fixed arrays, dynamic arrays, associative arrays and queues can be declared as rand or randc. All their elements are treated as random. Individual array elements can also be constrained, in this case, index expression must be constant. For dynamic arrays, the size of the array length can be constrained. Non integer data types like shortreal, real and realtime are not allowed for random

variable declaration.

### **Rand Modifier :**

Variables declared with rand keyword are standard random variables. When there are no other control on distribution, these variables are uniformly distributed across valid values.

#### **EXAMPLE:**

```
class rand_cl;
    rand bit [0:2] Var;
    constraint limit_c { Var < 4;}
endclass

program rand_p_9;
    rand_cl obj;
    integer count_0, count_1, count_2, count_3;
    initial
    begin
        obj = new();
        count_0 = 0; count_1 = 0; count_2 = 0; count_3 = 0;
        repeat(100000)
            begin
                void'(obj.randomize());
                if( obj.Var == 0) count_0++;
                else if( obj.Var == 1) count_1++;
                else if( obj.Var == 2) count_2++;
                else if( obj.Var == 3) count_3++;
            end
            $display(" count_0 = %0d , count_1 = %0d, count_2 = %0d, count_3 = %0d
",count_0, count_1, count_2, count_3);
        end
    endprogram
```

#### **RESULTS:**

count\_0 = 25046 , count\_1 = 24940, count\_2 = 24969, count\_3 = 25045

Simulation results show that the rand variable is distributed uniformly.

### **Randc Modifier :**

Variables declared as randc are random cyclic that randomly iterates over all the values in the range and no value is repeated within an iteration until every possible value has been assigned. But iteration sequences are won't be same. Bit and enumerated types can be randc variables. To reduce memory requirements, implementations can impose a limit on maximum size of a randc variable, it should be not be more than 8 bits.

**EXAMPLE:**

```
class rand_c;
  randc bit [1:0] Var;
endclass

program rand_cp_10;
rand_c obj=new();
initial
  for(int i=0;i<20;i++)
begin
  void'(obj.randomize());
  $write("%0d ",obj.Var);
  if(i%4==3)
    $display("");
end
endprogram
```

**RESULTS:**

```
# 0_3_1_2_
# 3_0_2_1_
# 0_3_1_2_
# 0_1_2_3_
# 3_0_2_1_
```

The permutation sequence for any given randc variable is recomputed whenever the constraints changes on that variable or when none of the remaining values in the permutation can satisfy the constraints.

**EXAMPLE:**

```
class rand_c;
  randc bit [2:0] Var;
  integer MIN = 4;
  constraint C { Var < MIN ;}
endclass
```

```
program rand_cp_11;
rand_c obj=new();
initial
  for(int i=0;i<20;i++)
begin
  obj.MIN = 4;
  if(i>12)
    obj.MIN=7;
```

```

void'(obj.randomize());
if(i==12)
    $display(" CONSTRAINT CHANGED ");
$write("%0d_",obj.Var);
if((i%4==3))
    $display("");
end
endprogram

```

**RESULTS:**

```

0_2_3_1_
0_1_3_2_
3_2_0_1_
CONSTRAINT CHANGED
0_1_4_2_
6_5_3_0_

```

Permutation sequence is computed on every call of new() function. So if randc variables won't behave like random cyclic, if new() is called for every randomization. In the following example variable Var is not behaving like random cyclic.

**EXAMPLE:**

```

class rand_c;
    randc bit [1:0]Var;
endclass
program rand_cp_12;
    rand_c obj=new();
    initial
        for(int i=0;i<20;i++)
        begin
            obj=new();
            void'(obj.randomize());
            $write("%0d_",obj.Var);
            if(i%4==3)
                $display("");
        end
endprogram

```

**RESULTS:**

```

# 1_3_1_2_
# 3_2_2_1_
# 2_0_0_0_
# 3_3_1_0_
# 3_0_1_0_

```

## **RANDOMIZATION METHODS**

### **Randomization Built-In Methods**

SystemVerilog has randomize(),pre\_randomize() and post\_randomize() built-in functions for randomization. Calling randomize() causes new values to be selected for all of the random variables in an object. To perform operations immediately before or after randomization,pre\_randomize() and post\_randomize() are used.

#### **Randomize()**

Every class has a virtual predefined function randomize(), which is provided for generating a new value. Randomization function returns 1 if the solver finds a valid solution. We cannot override this predefined function. It is strongly recommended to check the return value of randomize function. Constraint solver never fails after one successful randomization, if solution space is not changed. For every randomization call, check the return value, solver may fail due to dynamically changing the constraints. In the following example, there is no solution for Var < 100 and Var > 200, so the randomization fails.

The best way to check status of randomization return value is by using assertion.

```
assert(obj.randomize());
```

#### **EXAMPLE:**

```
program Simple_pro_13;
  class Simple;
    rand integer Var;
    constraint c1 { Var <100;};
    constraint c2 { Var >200;};
  endclass
  initial
  begin
    Simple obj = new();
    if(obj.randomize())
      $display(" Randomization sucessfull : Var = %0d ",obj.Var);
    else
      $display("Randomization failed");
  end
endprogram
```

#### **RESULTS:**

```
# Randomization failed
```

If randomize() fails, the constraints are infeasible and the random variables retain their previous values. In the following example, For the first randomization call there is a solution. When the constraints are changed, the randomization failed. Simulation results show that after randomization failed, random variables hold their previous values.

## EXAMPLE:

```
program Simple_pro_14;
  class Simple;
    rand integer Var;
    integer MIN = 20 ;
    constraint c { Var < 100 ; Var > MIN ; }
  endclass
  Simple obj;
  initial
  begin
    obj = new();
    if(obj.randomize())
      $display(" Randomization successful : Var = %0d ",obj.Var);
    else
      $display("Randomization failed: Var = %0d ",obj.Var);
    obj.MIN = 200;
    $display(" MIN is changed to fail the constraint");
    if(obj.randomize())
      $display(" Randomization sucessfull : Var = %0d ",obj.Var);
    else
      $display(" Randomization failed : Var = %0d",obj.Var);
  end
endprogram
```

## RESULTS:

```
# Randomization sucessfull : Var = 87
# MIN is changed to fail the constraint.
# Randomization failed : Var = 87
```

### Pre\_randomize And Post\_randomize

Every class contains pre\_randomize() and post\_randomize() methods, which are automatically called by randomize() before and after computing new random values. When randomize() is called, it first invokes the pre\_randomize() then randomize() finally if the randomization is successful only post\_randomize is invoked.

These methods can be used as hooks for the user to perform operations such as setting initial values and performing functions after assigning random variables.

## EXAMPLE:

```
program pre_post_15;
  class simple;
    function void pre_randomize;
      $display(" PRE_RANDOMIZATION ");
    endfunction
```

```

function void post_randomize;
    $display(" POST_RANDOMIZATION ");
endfunction
endclass
simple obj = new();
initial
    void'(obj.randomize());
endprogram

```

#### RESULTS:

```

# PRE_RANDOMIZATION
# POST_RANDOMIZATION

```

Overriding of pre\_randomize and post\_randomize functions is allowed by child class. If parent class functions are not called when overriding pre\_randomize() and post\_randomize functions, parent class function definitions will be omitted.

#### EXAMPLE:

```

class Base;
    function void pre_randomize;
        $display(" BASE PRE_RANDOMIZATION ");
    endfunction
    function void post_randomize;
        $display(" BASE POST_RANDOMIZATION ");
    endfunction
endclass
    class Extend_1 extends Base;
        function void pre_randomize;
            $display(" EXTEND_1 PRE_RANDOMIZATION ");
        endfunction
        function void post_randomize;
            $display(" EXTEND_1 POST_RANDOMIZATION ");
        endfunction
endclass
class Extend_2 extends Base;
    function void pre_randomize;
        super.pre_randomize();
        $display(" EXTEND_2 PRE_RANDOMIZATION ");
    endfunction
    function void post_randomize;
        super.post_randomize();
        $display(" EXTEND_2 POST_RANDOMIZATION ");
    endfunction

```

```

endclass
program pre_post_16;
  Base B = new();
  Extend_1 E1 = new();
  Extend_2 E2 = new();
  initial
  begin
    void'(B.randomize());
    void'(E1.randomize());
    void'(E2.randomize());
  end
endprogram

```

In the extended class EXTEND\_1, when overriding the builtin functions, parent class functions are not called. In the extended class EXTEND\_2, super.methods are called which invokes the parent class methods also.

## RESULTS:

```

# BASE PRE_RANDOMIZATION
# BASE POST_RANDOMIZATION
# EXTEND_1 PRE_RANDOMIZATION
# EXTEND_1 POST_RANDOMIZATION
# BASE PRE_RANDOMIZATION
# EXTEND_2 PRE_RANDOMIZATION
# BASE POST_RANDOMIZATION
# EXTEND_2 POST_RANDOMIZATION

```

The pre\_randomize() and post\_randomize() methods are not virtual. However, because they are automatically called by the randomize() method, which is virtual, they appear to behave as virtual methods. This example demonstrates that these functions are not virtual but simulation results show that, it executed extended class definition functions. Extended class object is created and assigned to base class object. Calls to pre\_randomize and post\_randomize calls in object B ,executed the extended class definitions.

## EXAMPLE:

```

class Base;
  function void pre_randomize;
    $display(" BASE PRE_RANDOMIZATION ");
  endfunction
  virtual function void post_randomize;
    $display(" BASE POST_RANDOMIZATION ");
  endfunction
endclass

```

```

class Extend extends Base;
function void pre_randomize;
    $display(" EXTEND PRE_RANDOMIZATION ");
endfunction
function void post_randomize;
    $display(" EXTEND POST_RANDOMIZATION ");
endfunction
endclass
program pre_post_17;
Base B ;
Extend E = new();
initial
begin
    B = E ;
    void'(B.randomize());
    void'(E.randomize());
end
endprogram

```

#### RESULTS:

There should be compilation error.

In the above example compilation error is due to the declaration of post\_randomize() function as virtual. By removing the virtual keyword for the post\_randomize() function, calling the randomize() function by parent and child class, both will execute functions of child class only. Which is a virtual function behaviour.

#### EXAMPLE:

```

class Base;
function void pre_randomize;
    $display(" BASE PRE_RANDOMIZATION ");
endfunction
function void post_randomize;
    $display(" BASE POST_RANDOMIZATION ");
endfunction
endclass

```

```

class Extend extends Base;
function void pre_randomize;
    $display(" EXTEND PRE_RANDOMIZATION ");
endfunction
function void post_randomize;

```

```

$display(" EXTEND POST_RANDOMIZATION ");
endfunction
endclass
program pre_post_17;
Base B ;
Extend E = new();
initial
begin
    B = E ;
    void'(B.randomize());
    void'(E.randomize());
end
endprogram

```

#### RESULTS:

```

# EXTEND PRE_RANDOMIZATION
# EXTEND POST_RANDOMIZATION
# EXTEND PRE_RANDOMIZATION
# EXTEND POST_RANDOMIZATION

```

If the class is a derived class and no user-defined implementation of pre\_randomize() and post\_randomize() exists, then pre\_randomize() and post\_randomize() will automatically invoke super.pre\_randomize() and super.post\_randomize() respectively.

#### EXAMPLE:

```

class Base;
function void pre_randomize;
    $display(" BASE PRE_RANDOMIZATION ");
endfunction
function void post_randomize;
    $display(" BASE POST_RANDOMIZATION ");
endfunction
endclass
class Extend extends Base;
endclass
program pre_post_19;
Extend E = new();
initial
    void'(E.randomize());
endprogram

```

**RESULTS:**

```
# BASE PRE_RANDOMIZATION  
# BASE POST_RANDOMIZATION
```

**EXAMPLE:**

```
class Base;  
    function void pre_randomize;  
        $display(" BASE PRE_RANDOMIZATION \n");  
    endfunction  
    function void post_randomize;  
        $display(" BASE POST_RANDOMIZATION \n");  
    endfunction  
endclass  
  
class Extend extends Base;  
    function void pre_randomize;  
        super.pre_randomize();  
        $display(" EXTENDED PRE_RANDOMIZATION \n");  
    endfunction  
    function void post_randomize;  
        $display(" EXTENDED POST_RANDOMIZATION \n");  
    endfunction  
endclass  
  
program pre_post;  
    Base B;  
    Extend E = new();  
    initial  
    begin  
        B = E;  
        if(B.randomize())  
            $display(" randomization done \n");  
    end  
endprogram
```

**RESULTS:**

```
BASE PRE_RANDOMIZATION  
EXTENDED PRE_RANDOMIZATION  
EXTENDED POST_RANDOMIZATION  
randomization done
```

Results show that, if extended class is having new definition, explicitly super.pre\_ or post\_ has to be called.

super.pre\_randomize() is called in extended class, but super.post\_randomize() is not called in above example. See the difference in results.

If a class A instance is in Class B, To randomize class A by calling the randomize function of class B, Class A instance has to be declared as rand variable.

**EXAMPLE:**

```
class A;  
rand integer Var;  
endclass  
  
class B;  
rand A obj_1 = new();  
A obj_2 = new();  
endclass  
  
program a_b_20;  
B obj=new();  
initial  
begin  
obj.obj_1.Var = 1;  
obj.obj_2.Var = 1;  
repeat(10)  
begin  
void'(obj.randomize());  
$display(" Var1 = %d ,Var2 = %d ",obj,obj_1.Var,obj,obj_2.Var );  
end  
end  
endprogram
```

**RESULTS:**

```
# Var1 = 733126180 ,Var2 = 1  
# Var1 = -119008195 ,Var2 = 1  
# Var1 = 342785185 ,Var2 = 1  
# Var1 = 679818185 ,Var2 = 1  
# Var1 = -717162992 ,Var2 = 1  
# Var1 = 664520634 ,Var2 = 1  
# Var1 = -1944980214 ,Var2 = 1  
# Var1 = -1350759145 ,Var2 = 1
```

```
# Var1 = -1374963034 ,Var2 = 1  
# Var1 = -462078992 ,Var2 = 1
```

Look at the results. Variable of obj\_2 is not randomized. Only variable of obj\_1 which is declared as rand is randomized.

Upon calling the randomize method of B object which contains rand A object, First B prerandomize is called, then A prerandomize method is called, then B is randomized, if a solution was found, new values are assigned to the random A objects. If solution was found, for each random object that is a class instance it's post\_randomize method is called. That means if randomization is successful next B postrandomize, next A postrandomize functions are called. Upon calling B randomize function this is sequence it follow.

B-Prerandomize --> A-prerandomize --> A.randomize --> B-postrandomize --> A-postrandomize

**EXAMPLE:**

```
class A;  
rand integer Var;  
function void pre_randomize;  
    $display(" A PRE_RANDOMIZATION ");  
endfunction  
function void post_randomize;  
    $display(" A POST_RANDOMIZATION ");  
endfunction  
endclass  
  
class B;  
rand A obj_a;  
function new();  
    obj_a = new();  
endfunction  
function void pre_randomize;  
    $display(" B PRE_RANDOMIZATION ");  
endfunction  
function void post_randomize;  
    $display(" B POST_RANDOMIZATION ");  
endfunction  
endclass  
  
program pre_post_21;  
B obj_b = new();  
initial  
    void'(obj_b.randomize());
```

**endprogram**

**RESULTS:**

```
# B PRE_RANDOMIZATION  
# A PRE_RANDOMIZATION  
# B POST_RANDOMIZATION  
# A POST_RANDOMIZATION
```

If randomization failed for obj\_a, then post\_randomize of obj\_a and post\_randomize of obj\_b won't be called, and randomization will fail for obj\_b also.

**EXAMPLE:**

```
class A;  
    rand bit [2:0] Var;  
    constraint randge_c { Var > 2 ; Var < 2; }  
    function void pre_randomize;  
        $display(" A PRE_RANDOMIZATION ");  
    endfunction  
    function void post_randomize;  
        $display(" A POST_RANDOMIZATION ");  
    endfunction  
endclass  
  
class B;  
    rand A obj_a;  
    function void pre_randomize;  
        $display(" B PRE_RANDOMIZATION ");  
    endfunction  
    function void post_randomize;  
        $display(" B POST_RANDOMIZATION ");  
    endfunction  
    function new();  
        obj_a = new();  
    endfunction  
endclass  
program pre_post_22;  
    B obj_b = new();  
    initial  
        void'(obj_b.randomize());  
endprogram
```

## **RESULTS:**

```
# B PRE_RANDOMIZATION  
# A PRE_RANDOMIZATION
```

### **Disabling Random Variable**

The random nature of variables declared as rand or randc can be turned on or off dynamically. To change the status of variable which is declared as rand or randc to state variable, built in rand\_mode() method is used. State variables are not randomized by randomize() method. By default all rand and randc variables are active. When called as a task, the arguments to the rand\_mode method determines the operation to be performed. If the argument is 0, then all the variables declared as rand and randc will become non random i.e all random variables treated as state variables. If argument is 1, then all variables declared as rand and randc will be randomized.

#### **EXAMPLE:**

```
class rand_mo;
```

```
    rand integer Var1;
```

```
    rand integer Var2;
```

```
endclass
```

```
program rand_mo_p_23;
```

```
    rand_mo obj = new();
```

```
    initial
```

```
    begin
```

```
        void'(obj.randomize());
```

```
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
```

```
        obj.rand_mode(0);          // Var1 and Var2 will be treated as State variables.
```

```
        void'(obj.randomize());
```

```
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
```

```
        obj.rand_mode(1);          // // Var1 and Var2 will be treated as random variables.
```

```
        void'(obj.randomize());
```

```
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
```

```
    end
```

```
endprogram
```

## **RESULTS:**

```
# Var1 : 733126180  Var2 : -119008195  
# Var1 : 733126180  Var2 : -119008195  
# Var1 : 342785185  Var2 : 679818185
```

If arguments are Variable name, then only that variable will be non random.

**EXAMPLE:**

```
class rand_mo;
    rand integer Var1;
    rand integer Var2;
endclass

program rand_mo_p_24;
rand_mo obj = new();
initial
begin
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    obj.Var1.rand_mode(0);      // Var1 will become State variable
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    obj.Var2.rand_mode(0);      // Var2 will also become State variable
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    obj.Var1.rand_mode(1);      // // Var1 will become random variable
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
end
endprogram
```

**RESULTS:**

```
# Var1 : 733126180  Var2 : -119008195
# Var1 : 733126180  Var2 : 342785185
# Var1 : 733126180  Var2 : 342785185
# Var1 : 679818185  Var2 : 342785185
```

When rand\_mode method is called as function, it returns the active status of the specified random variable.

**EXAMPLE:**

```
class rand_mo;
    rand integer Var1;
    rand integer Var2;
endclass

program rand_mo_p_24;
rand_mo obj = new();
initial
begin
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
```

```

obj.Var1.rand_mode(0);
void'(obj.randomize());
$display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
if(obj.Var1.rand_mode())
    $display(" Var1 is random");
else
    $display(" Var1 is nonrandom");
void'(obj.randomize());
$display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
end
endprogram

```

#### **RESULTS:**

```

# Var1 : 733126180  Var2 : -119008195
# Var1 : 733126180  Var2 : 342785185
# Var1 is nonrandom
# Var1 : 733126180  Var2 : 679818185

```

If you are changing the status of a variable, which is not existing it is compilation error.

#### **EXAMPLE:**

```

class rand_mo;
    rand integer Var1;
    rand integer Var2;
endclass

```

```

program rand_mo_p_24;
    rand_mo obj = new();
    initial
        begin
            void'(obj.randomize());
            $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
            obj.Var3.rand_mode(0);
            void'(obj.randomize());
            $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
            if(obj.Var3.rand_mode())
                $display(" Var3 is nonrandom");
            void'(obj.randomize());
            $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
        end
endprogram

```

A compiler error shall be issued if the specified variable does not exist within the class hierarchy or even though it exists but not declared as rand or randc. The following example illustrates the second case.

**EXAMPLE:**

```
class rand_mo;
    rand integer Var1;
    integer Var2;
endclass

program rand_mo_p_24;
rand_mo obj = new();
initial
begin
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    obj.Var2.rand_mode(0);
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    if(obj.Var2.rand_mode())
        $display(" Var1 is nonrandom");
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
end
endprogram
```

In the above example, Var2 is state variable. If the random variable is an object handle, only the mode of the object is changed, not the mode of random variables within that object.

**EXAMPLE:**

```
class rand_var;
    rand integer Var2;
endclass

class rand_mo;
    rand integer Var1;
    rand rand_var rv;
    function new();
        rv = new();
    endfunction
endclass

program rand_mo_p_23;
rand_mo obj = new();
initial
begin
```

```

void'(obj.randomize());
$display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.rv.Var2);
obj.rand_mode(0);
void'(obj.randomize());
$display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.rv.Var2);
void'(obj.rv.randomize());
$display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.rv.Var2);
end
endprogram

```

#### **RESULTS:**

```

Var1 : 345345423  Var2 : 234563556
Var1 : 345345423  Var2 : 234563556
Var1 : 345345423  Var2 : -2456456

```

#### **Random Static Variable**

Randomization does not depend on life time of variable. Even if a variable is static,randomization is specific to object.So rand\_mode() on static variable,only switches off the randomization on the variable of that object.This is true for dist and randc.

In the following example, Var1 is static and Var2 is automatic.Var1 and Var2 in obj\_2 are made nonrandom using rand\_mode(0).Var1 and Var2 in obj\_1 are getting randomized.The only difference between Var1 and Var2 is that new random value for Var1 is appreas on both objects.

#### **EXAMPLE:**

```

class A;
  rand static integer Var1;
  rand integer Var2;
endclass

program A_p_27;
A obj_1 = new;
A obj_2 = new;
  initial
begin
  obj_2.Var1.rand_mode(0);
  obj_2.Var2.rand_mode(0);
  repeat(2)
    begin
      void'(obj_1.randomize());
      void'(obj_2.randomize());
      $display("obj_1.Var1 : %d ,obj_1.Var2 : %d : obj_2.Var1 : %d ,obj_2.Var2 : %d
      :",obj_1.Var1,obj_1.Var2,obj_2.Var1,obj_2.Var2);
    end
end

```

**endprogram**

**RESULTS:**

obj\_1.Var1 : 934734534,obj\_1.Var2 : 234232342: obj\_2.Var1 : 934734534 ,obj\_2.Var2 : 0 :

obj\_1.Var1 : 908123314,obj\_1.Var2 : 121891223: obj\_2.Var1 : 908123314 ,obj\_2.Var2 : 0 :

Random variables declared as static are shared by all instances of the class in which they are declared. Each time the randomize() method is called, the variable is changed in every class instance.

**Randomizing Nonrand Variable**

All the variables(randc, rand and nonrandom variables) randomization nature can be changed dynamically. Using rand\_mode() rand and randc variables changes its nature. The random nature of variables which are not declared as rand or randc can also be randomized dynamically. When the randomize method is called with no arguments, it randomizes the variables which are declared as rand or randc, so that all of the constraints are satisfied. When randomize is called with arguments, those arguments designate the complete set of random variables within that object, all other variables in the object are considered state variables.

**EXAMPLE:**

```
class CA;
  rand byte x, y;
  byte v, w;
  constraint c1 { x < v && y > w ;}
endclass
```

```
program CA_p_28;
  CA a = new;
  initial
  begin
    a.x = 10;a.y = 10;a.v = 10;a.w = 10;
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize()); // random variables: x, y state variables: v, w
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize(x)); // random variables: x state variables: y, v, w
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize(v,w)); // random variables: v, w state variables: x, y
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize(w,x)); // random variables: w, x state variables: y, v
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
  end
endprogram
```

## RESULTS:

```
# x : 10 y : 10 : v : 10 : w : 10
# x : -71 y : 96 : v : 10 : w : 10
# x : -37 y : 96 : v : 10 : w : 10
# x : -37 y : 96 : v : -22 : w : 80
# x : -90 y : 96 : v : -22 : w : -41
```

In above example x and y are rand variables, v and w are state variables. When a.randomize() is called, all rand variables are randomized and state variables are hold the same value. When a.randomize(w) is called, only w is considered as rand variable and all others as state variables. Here w is in constraint block so it has to satisfy the constraints. v,y and x also are state variables now, they also need to satisfy the constraint else it fails.

Replacing the class variables, with its hierarchical result also should result same.

## EXAMPLE:

```
program CA_p_29;
CA a = new;
initial
begin
    a.x = 10;a.y = 10;a.v = 10;a.w = 10;
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize()); // random variables: x, y state variables: v, w
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize( a.x )); // random variables: x state variables: y, v, w
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize( a.v, a.w )); // random variables: v, w state variables: x, y
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
    void'(a.randomize( a.w, a.x )); // random variables: w, x state variables: y, v
    $display(" x : %3d y : %3d : v : %3d : w : %3d ",a.x,a.y,a.v,a.w);
end
endprogram
```

## RESULTS:

```
# x : 10 y : 10 : v : 10 : w : 10
# x : -71 y : 96 : v : 10 : w : 10
# x : -37 y : 96 : v : 10 : w : 10
# x : -37 y : 96 : v : -22 : w : 80
# x : -90 y : 96 : v : -22 : w : -41
```

If you are not interested to satisfy the constraints in constraint block, instead of switching off the constraint block, just randomize the variables using scope randomize() function. Scope randomize provide the ability to randomize class variables also along with non class variables.

**EXAMPLE:**

```
program CA_p_30;
    integer x,y,v,w;
    initial
    begin
        x = 10;y = 10;v = 10;w = 10;
        $display(" x : %3d y : %3d : v : %3d : w : %3d ",x,y,v,w);
        randomize( x ); // random variables: x state variables: y, v, w
        $display(" x : %3d y : %3d : v : %3d : w : %3d ",x,y,v,w);
        randomize(v,w ); // random variables: v, w state variables: x, y
        $display(" x : %3d y : %3d : v : %3d : w : %3d ",x,y,v,w);
        randomize(w,x ); // random variables: w, x state variables: y, v
        $display(" x : %3d y : %3d : v : %3d : w : %3d ",x,y,v,w);
    end
endprogram
```

**RESULTS:**

```
# x : 10 y : 10 : v : 10 : w : 10
# x : -826701341 y : 10 : v : 10 : w : 10
# x : -826701341 y : 10 : v : 541037099 : w : -457978538
# x : 978699393 y : 10 : v : 541037099 : w : -683182079
```

**CHECKER**

Built-in method randomize() not only used for randomization, it can be used as checker. When randomize() method is called by passing null, randomize() method behaves as checker instead of random generator. It evaluates all the constraints and returns the status. This is true for both scope randomization function and class randomization function. When a randomize() method is called, first RNG assigns values to random variables and then solver checks the constraints. When randomize(null) is called, it wont call the RNG to assign values to random variables, it just solves constraints.

**EXAMPLE:**

```
class Eth_rx;
    rand integer Pkt_len;
    rand integer Var;
    constraint var_c { Var < 1518 ;Var > 64 ;}
endclass
```

```

program Eth_25;
  Eth_rx rx = new();
  initial
  begin
    rx.Pkt_len = 32;
    rx.Var = 871;
    if(rx.randomize(null))
      $display(" VALID PKT IS RECEIVED ");
    else
      $display(" INVALID PKT IS RECEIVED ");
    end
  endprogram

```

### RESULTS:

# VALID PKT IS RECEIVED

Constraints can be written without having random variables in expressions. If there is any constraint on state variables and they are dynamically changed, and if you want to make sure that these dynamic changes should satisfy the constraint, use randomize check to make sure that relation is satisfied.

In the following example, MIN and MAX are dynamically controllable state variables.

Constraint checker\_c fails when MIN = 50 and MAX = 10.

### EXAMPLE:

```

class Base;
  rand integer Var;
  integer MIN,MAX;
  constraint randge_r { Var < MAX ; Var > MIN ;}
  constraint checker_c{ MIN < MAX ;} // This checks that these dynamic variables are valid
  task set (integer MIN,integer MAX);
    this.MIN = MIN;
    this.MAX = MAX;
    $display( " SET : MIN = %0d , MAX = %0d ",MIN,MAX);
  endtask
endclass

```

```

program inhe_26;
  Base obj;
  initial
  begin
    obj = new();
    obj.set(0,100) ;
    for(int i=0 ; i < 5 ; i++)

```

```

if(obj.randomize())
    $display(" Randomization sucessfull : Var = %0d ",obj.Var);
else
    $display("Randomization failed");

obj.set(50,10) ;
for(int i=0 ; i < 5 ; i++)
    if(obj.randomize())
        $display(" Randomization sucessfull : Var = %0d ",obj.Var);
    else
        $display("Randomization failed");
end
endprogram

```

#### **RESULTS:**

```

# SET : MIN = 0 , MAX = 100
# Randomization sucessfull : Var = 68
# Randomization sucessfull : Var = 11
# Randomization sucessfull : Var = 8
# Randomization sucessfull : Var = 36
# Randomization sucessfull : Var = 64
# SET : MIN = 50 , MAX = 10
# Randomization failed

```

#### **CONSTRAINT BLOCK**

Constraint block contains declarative statements which restrict the range of variable or defines the relation between variables. Constraint programming is a powerful method that lets users build generic, reusable objects that can be extended or more constrained later. constraint solver can only support 2 state values. If a 4 state variable is used, solver treats them as 2 state variable.. Constraint solver fails only if there is no solution which satisfies all the constraints. Constraint block can also have nonrandom variables, but at least one random variable is needed for randomization. Constraints are tied to objects. This allows inheritance, hierarchical constraints, controlling the constraints of specific object.

#### **Inheritance**

One of the main advantage of class randomization is Inheritance. Constraints in derived class with the same name in base class overrides the base class constraints just like task and functions.

### EXAMPLE:

```
class Base;
    rand integer Var;
    constraint range { Var < 100 ; Var > 0 ;}
endclass

class Extended extends Base;
    constraint range { Var < 100 ; Var > 50 ;} // Overriding the Base class constraints.
endclass

program inhe_31;
    Extended obj;
    initial
    begin
        obj = new();
        for(int i=0 ; i < 100 ; i++)
            if(obj.randomize())
                $display(" Randomization sucessfull : Var = %0d ",obj.Var);
            else
                $display("Randomization failed");
    end
endprogram
```

### RESULTS:

```
# Randomization sucessfull : Var = 91
# Randomization sucessfull : Var = 93
# Randomization sucessfull : Var = 77
# Randomization sucessfull : Var = 68
# Randomization sucessfull : Var = 67
# Randomization sucessfull : Var = 52
# Randomization sucessfull : Var = 71
# Randomization sucessfull : Var = 98
# Randomization sucessfull : Var = 69
```

Adding new constraints in the derived class, can change the solution space. Solver has to solve both constraints defined in base class and derived class. In the example given below, Constraint range\_1 defines the range that Var is between 0 to 100. Constraint range\_2 limits the Var to be greater than 50 and solver has to solve both the constraints and the solution space is between 50 to 100.

### EXAMPLE:

```
class Base;
    rand integer Var;
    constraint range_1 { Var < 100 ; Var > 0 ;}
endclass
```

```

class Extended extends Base;
  constraint range_2 { Var > 50 ;} // Adding new constraints in the Extended class
endclass

program inhe_32;
  Extended obj;
  initial
  begin
    obj = new();
    for(int i=0 ; i < 20 ; i++)
      if(obj.randomize())
        $write(": Var = %0d : ",obj.Var);
      else
        $display("Randomization failed");
  end
endprogram

```

#### RESULTS:

Var = 91 :: Var = 93 :: Var = 77 :: Var = 68 :: Var = 67 :: Var = 52 :: Var = 71 :: Var = 98 :: Var = 69 :: Var = 70 :: Var = 96 :: Var = 88 :: Var = 84 :: Var = 99 :: Var = 68 :: Var = 83 :: Var = 52 :: Var = 72 :: Var = 93 :: Var = 80 :

#### Overriding Constraints

The randomize() task is virtual. Accordingly it treats the class constraints in a virtual manner. When a named constraint is redefined in an extended class, the previous definition is overridden and when casting extended class to base class does not change the constraint set.

#### EXAMPLE:

```

class Base;
  rand integer Var;
  constraint range { Var < 100 ; Var > 0 ;}
endclass

class Extended extends Base;
  constraint range { Var == 100 ;} // Overriding the Base class constraints.
endclass

program inhe_33;
  Extended obj_e;
  Base obj_b;
  initial
  begin
    obj_e = new();
    obj_b = obj_e;
    for(int i=0 ; i < 7 ; i++)

```

```

if(obj_b.randomize())
    $display(" Randomization sucessfull : Var = %0d ",obj_b.Var);
else
    $display("Randomization failed");
end
endprogram

```

#### RESULTS:

```

# Randomization sucessfull : Var = 100

```

When an extended object is casted to base object, all the constraints in extended object are solved along with the constraints in base object.

#### EXAMPLE:

```

class Base;
rand integer Var;
constraint range_1 { Var < 100 ; Var > 0 ;}
endclass

class Extended extends Base;
constraint range_2 { Var > 50 ;}
endclass

program inhe_34;
Extended obj_e;
Base obj_b;
initial
begin
    obj_e = new();
    obj_b = obj_e;
    for(int i=0 ; i < 10 ; i++)
        if(obj_b.randomize())
            $display(" Randomization sucessfull : Var = %0d ",obj_b.Var);
        else
            $display("Randomization failed");
end
endprogram

```

#### RESULTS:

```

# Randomization sucessfull : Var = 91
# Randomization sucessfull : Var = 93
# Randomization sucessfull : Var = 77
# Randomization sucessfull : Var = 68
# Randomization sucessfull : Var = 67
# Randomization sucessfull : Var = 52
# Randomization sucessfull : Var = 71
# Randomization sucessfull : Var = 98
# Randomization sucessfull : Var = 69
# Randomization sucessfull : Var = 70

```

### **INLINE CONSTRAINT**

Inline constraints allows to add extra constraints to already existing constraints which are declared inside class. If you have constraints already defined for variavle var, solver solves those constraints along with the in-line constraints.

#### **EXAMPLE:**

```

class inline;
  rand integer Var;
  constraint default_c { Var > 0 ; Var < 100;}
endclass

program inline_p_35;
  inline obj;
  initial
  begin
    obj = new();
    repeat(5)
      if(obj.randomize() with { Var == 50;})
        $display(" Randodmize sucessful Var %d ",obj.Var);
      else
        $display(" Randomization failes");
    end
endprogram

```

#### **RESULTS:**

```

# Randodmize sucessful Var      50

```

In the above example, by default only default\_c constraints are considered. Using inline constraint Var == 50 resulted value on variable Var based on both the default\_c and inline constraints. The scope for variable names in a constraint block, from inner to outer, is

randomize()...with object class, automatic and local variables, task and function arguments, class variables, and variables in the enclosing scope. The randomize()...with class is brought into scope at the innermost nesting level. In the f.randomize() with constraint block, x is a member of class Foo and hides the x in program Bar. It also hides the x argument in the doit() task. y is a member of Bar. z is a local argument.

In the example below, the randomize()...with class is Foo.

**EXAMPLE:**

```

class Foo;
  rand integer x;
endclass
program Bar_36;
  Foo obj = new();
  integer x;
  integer y;
  task doit(Foo f, integer x, integer z);
    int result;
    result = f.randomize() with { x < y + z; };
    $display(":: obj.x : %d :: x : %d :: y : %d :: z : %d ::",obj.x,x,y,z);
  endtask
  initial
  begin
    x = 'd10;
    repeat(5)
      begin
        y = $urandom % 10;
        doit(obj,x ,'d12);
      end
    end
  endprogram

```

**RESULTS:**

:: obj.x : -1538701546 :: x :	10 :: y :	5 :: z :	12 ::
:: obj.x : -1048494686 :: x :	10 :: y :	9 :: z :	12 ::
:: obj.x : -1122673684 :: x :	10 :: y :	8 :: z :	12 ::
:: obj.x : -2050360121 :: x :	10 :: y :	7 :: z :	12 ::
:: obj.x : -886228933 :: x :	10 :: y :	3 :: z :	12 ::

By seeing this output we can tell the variable which used in inline constraint is class member x. constraint { x < y + z } means { obj.x < Bar\_36.y + do\_it.z }

## **GLOBAL CONSTRAINT**

SystemVerilog allows to have constraints between variables of different objects. These are called global constraints. Using the hierarchy notation, constraints can be applied on variables of different objects. When object is randomized, so are the contained objects and all other constraints are considered simultaneously.

### **EXAMPLE:**

```
class child;
    rand int Var1;
endclass

class parent;
    rand child child_obj;
    rand int Var2;
    constraint global_c { Var2 < child_obj.Var1 ;}
    function new();
        child_obj = new();
    endfunction
endclass

program random_37;
initial
    for(int i=0;i<5;i++)
begin
    parent parent_obj;
    parent_obj = new ();
    void'(parent_obj.randomize ());
    $display(" Var1 = %0d ,Var2 = %0d ",parent_obj.child_obj.Var1,parent_obj.Var2 );
end
endprogram
```

### **RESULTS:**

```
# Var1 = 903271284 ,Var2 = -1102515531
# Var1 = 2112727279 ,Var2 = -838916208
# Var1 = 1614679637 ,Var2 = 1572451945
# Var1 = 1284140376 ,Var2 = -373511538
# Var1 = 463675676 ,Var2 = -516850003
```

## **CONSTRAINT MODE**

### **Disabling Constraint Block**

SystemVerilog supports to change the status of constraint block dynamically. To change the status of a Constraint block, built in constraint\_mode() method is used. By default all the constraint blocks are active. When it is called as task, the arguments to the task determines the operation to be performed. If the arguments are 0 or 1, then all the constraints blocks are effected.

**EXAMPLE:**

```
class rand_mo;
    rand integer Var1;
    rand integer Var2;
    constraint Var_1 { Var1 == 20; }
    constraint Var_2 { Var2 == 10; }
endclass

program rand_mo_p_38;
    rand_mo obj = new();
    initial
    begin
        void'(obj.randomize());           //By default all constraints are active.
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
        obj.constraint_mode(0);          //Both constraints Var_1 and Var_2 are turned off.
        void'(obj.randomize());
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
        obj.constraint_mode(1);          //Both constraints Var_1 and Var_2 are turned on.
        void'(obj.randomize());
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    end
endprogram
```

**RESULTS:**

```
# Var1 :      20  Var2 :      10
# Var1 :  733126180  Var2 : -119008195
# Var1 :      20  Var2 :      10
```

If the arguments are Variable name, then only the status of that constraint block is changed.

**EXAMPLE:**

```
class rand_mo;
    rand integer Var1;
    rand integer Var2;
    constraint Var_1 { Var1 == 20; }
    constraint Var_2 { Var2 == 10; }
endclass

program rand_mo_p_38;
    rand_mo obj = new();
    initial
    begin
        void'(obj.randomize());
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
        obj.Var_1.constraint_mode(0);
```

```

void'(obj.randomize());
$display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
obj.Var_1.constraint_mode(1);
void'(obj.randomize());
$display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
end
endprogram

```

#### RESULTS:

```

Var1 : 20  Var2 : 10
Var1 : 3w456456  Var2 : 10
Var1 : 20  Var2 : 10

```

When it is called as function, it returns the active status of the specified constraint block.

#### EXAMPLE:

```

class rand_mo;
  rand integer Var1;
  rand integer Var2;
  constraint Var_1 { Var1 == 20; }
  constraint Var_2 { Var2 == 10; }
endclass

program rand_mo_p_38;
  rand_mo obj = new();
  initial
  begin
    void'(obj.randomize());           //By default all constraints are active.
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    obj.Var_1.constraint_mode(0);      //Both constraint Var_1 is turned off.
    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
    if (obj.Var_1.constraint_mode())
      $display("Var_1 constraint si active");
    else
      $display("Var_1 constraint si inactive");

    if (obj.Var_2.constraint_mode())
      $display("Var_2 constraint si active");
    else
      $display("Var_2 constraint si inactive");

    void'(obj.randomize());
    $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);
  
```

```
    end  
endprogram
```

#### RESULTS:

```
Var1 :      20  Var2 :      10  
Var1 : -2048772810  Var2 :      10  
Var_1 constraint si inactive  
Var_2 constraint si active  
Var1 : -673275588  Var2 :      10
```

If you are changing the status of a constraint block, which is not existing, then there should be a compilation error else contact your Application engineer to file the bug.

#### EXAMPLE:

```
class rand_mo;  
    rand integer Var1;  
    rand integer Var2;  
    constraint Var_1 { Var1 == 20; }  
    constraint Var_2 { Var2 == 10; }  
endclass  
program rand_mo_p_38;  
    rand_mo obj = new();  
    initial  
    begin  
        void'(obj.randomize());  
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);  
        obj.Var_3.constraint_mode(0);  
        void'(obj.randomize());  
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);  
        if(obj.Var_3.constraint_mode())  
            $display(" Var_1 constraint block is off");  
        void'(obj.randomize());  
        $display(" Var1 : %d  Var2 : %d ",obj.Var1,obj.Var2);  
    end  
endprogram
```

### EXTERNAL CONSTRAINTS

Constraint blocks can be defined externally in the same file or other files. Defining the constraints in external file, gives some what aspect oriented style of programming.

For example, test\_1 requires Var between 0 to 100 and tets\_2 requires Var between 50 to 100. Declare the constraint block as empty and define them in other files.

**EXAMPLE:**

```
class Base;
    rand integer Var;
    constraint range;
endclass

program inhe_39;
Base obj;
initial
begin
    obj = new();
    for(int i=0 ; i < 100 ; i++)
        if(obj.randomize())
            $display(" Randomization sucessfull : Var = %0d ",obj.Var);
        else
            $display("Randomization failed");
end
endprogram
```

In test\_1 file,**include**

```
constraint Base::range { Var < 100; Var > 0;}
```

In test\_2 file,**include**

```
constraint Base::range { Var < 100; Var > 50;}
```

Very popular verilog style of including testcases is by using `include which can also be used here.

Write the constraints in a file and include it.

//

**EXAMPLE:**

```
class Base;
    rand integer Var;
`include "constraint.sv"
endclass

program inhe_40;
Base obj;
initial
begin
    obj = new();
    for(int i=0 ; i < 100 ; i++)
```

```

if(obj.randomize())
    $display(" Randomization sucessfull : Var = %0d ",obj.Var);
else
    $display("Randomization failed");
end
endprogram

```

### **Constraint Hiding**

In SV Std 1800-2005 LRM , its not mentioned any where that constraints can be declared as local or protected. If they support to declare constraints as local, it would be helpful not to switch off the constraint block accidentally as it is not supposed to be done. The constraint BNF explicitly excludes the local and protected modifiers. The main reason for their exclusion is because constraints behave like virtual methods that are called by the built-in randomize method. If a constraint were declared local/protected it would still be visible to randomize and participate in the constraint equations. The only limitation would be to call the constraint\_mode on local/protected constraints from certain methods, and this does not seem very useful and probably create more confusion with regards to overridden methods.

### **RANDOMIZATION CONTROLLABILITY**

#### **Controlability**

Additional to the controllability features supported by SystemVerilog, following are more points with which controlability can be achieved.

In the following example, MACROS MIN\_D and MAX\_D are defined. Set the MIN and MAX values in the pre\_randomize as shown. As MIN\_D and MAX\_D are macros, they can be assigned from command line. Biggest disadvantage for the method shown below is dynamic controllability.

#### **EXAMPLE:**

```

`define MAX_D 100
`define MIN_D 50
class Base;
    rand integer Var;
    integer MIN,MAX;
    constraint randge { Var < MAX ; Var > MIN ;}
    function void pre_randomize ();
        this.MIN = `MIN_D;
        this.MAX = `MAX_D;
        $display( " PRE_RANDOMIZE : MIN = %0d , MAX = %0d ",MIN,MAX);
    endfunction
endclass
program inhe_42;
    Base obj;
    initial

```

```

begin
    obj = new();
    for(int i=0 ; i < 100 ; i++)
        if(obj.randomize())
            $display(" Randomization sucessfull : Var = %0d ",obj.Var);
        else
            $display("Randomization failed");
    end
endprogram

```

### RESULTS:

```

# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 91
# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 93
# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 77
# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 68
# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 67
# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 52
# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 71
# PRE_RANDOMIZE : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 98
...etc.

```

As in this example,a single object is created and randomized 100 times. Due to this,pre\_reandomize is called 100 times, which may not be preferred.

By assigning the values while declaration itself this can be avoided. Simpler way to achieve the above logic.

### EXAMPLE:

```

`define MAX_D 100
`define MIN_D 50
class Base;
    rand integer Var;
    integer MIN = `MIN_D;
    integer MAX = `MAX_D;
    constraint range { Var < MAX ; Var > MIN ;}
endclass

```

```

program inhe_43;
  Base obj;
  initial
  begin
    obj = new();
    for(int i=0 ; i < 100 ; i++)
      if(obj.randomize())
        $display(" Randomization sucessfull : Var = %0d ",obj.Var);
      else
        $display("Randomization failed");
  end
endprogram

```

#### RESULTS:

```

# Randomization sucessfull : Var = 91
# Randomization sucessfull : Var = 93
# Randomization sucessfull : Var = 77
# Randomization sucessfull : Var = 68
# Randomization sucessfull : Var = 67
# Randomization sucessfull : Var = 52
# Randomization sucessfull : Var = 71
# Randomization sucessfull : Var = 98
# Randomization sucessfull : Var = 69
# Randomization sucessfull : Var = 70
...etc.

```

With the above approach also, dynamic controllability is lost. For dynamic controllability, define a task, pass this values as arguments when ever the changed is needed.

#### EXAMPLE:

```

class Base;
  rand integer Var;
  integer MIN = 10,MAX = 20; // Define default values,If function set is not called,with this it
  will work
  constraint randge { Var < MAX ; Var > MIN ;}
  task set (integer MIN,integer MAX);
    this.MIN = MIN;
    this.MAX = MAX;
    $display( " SET : MIN = %0d , MAX = %0d ",MIN,MAX);
  endtask
endclass

```

```

program inhe_44;

```

```

Base obj;
initial
begin
    obj = new();
    obj.set(0,100) ;
    for(int i=0 ; i < 5 ; i++)
        if(obj.randomize())
            $display(" Randomization sucessfull : Var = %0d ",obj.Var);
        else
            $display("Randomization failed");
    obj.set(50,100) ;
    for(int i=0 ; i < 5 ; i++)
        if(obj.randomize())
            $display(" Randomization sucessfull : Var = %0d ",obj.Var);
        else
            $display("Randomization failed");
end

```

**endprogram**

**RESULTS:**

```

# SET : MIN = 0 , MAX = 100
# Randomization sucessfull : Var = 24
# Randomization sucessfull : Var = 68
# Randomization sucessfull : Var = 43
# Randomization sucessfull : Var = 11
# Randomization sucessfull : Var = 4
# SET : MIN = 50 , MAX = 100
# Randomization sucessfull : Var = 52
# Randomization sucessfull : Var = 71
# Randomization sucessfull : Var = 98
# Randomization sucessfull : Var = 69
# Randomization sucessfull : Var = 70

```

More simpler way to dynamically modifying the constraints is by modifying the data members of class via object reference.

**EXAMPLE:**

```

class Base;
    rand integer Var;
    integer MIN = 20,MAX =30;
    constraint randge { Var < MAX ; Var > MIN ;}
endclass
program inhe_45;

```

```

Base obj;
initial
begin
    obj = new();
    obj.MIN = 0;
    obj.MAX = 100;
    for(int i=0 ; i < 5 ; i++)
        if(obj.randomize())
            $display(" Randomization sucessfull : Var = %0d ",obj.Var);
    else
        $display("Randomization failed");
        $display("MIN and MAX changed");
    obj.MIN = 50;
    obj.MAX = 100;
    for(int i=0 ; i < 5 ; i++)
        if(obj.randomize())
            $display(" Randomization sucessfull : Var = %0d ",obj.Var);
    else
        $display("Randomization failed");
end
endprogram

```

#### **RESULTS:**

```

# Randomization sucessfull : Var = 24
# Randomization sucessfull : Var = 68
# Randomization sucessfull : Var = 43
# Randomization sucessfull : Var = 11
# Randomization sucessfull : Var = 4
# MIN and MAX changed
# Randomization sucessfull : Var = 52
# Randomization sucessfull : Var = 71
# Randomization sucessfull : Var = 98
# Randomization sucessfull : Var = 69
# Randomization sucessfull : Var = 70

```

#### **STATIC CONSTRAINT**

If a constraint block is declared as static, then constraint mode on that block will effect on all the instances of that class. In the following example, two objects obj\_1 and obj\_2 are declared. Constraints Var1\_c is static and Var2\_c is not static. When constraint\_mode is used to switch off the constraints in obj\_2, constraint var1\_c in obj\_1 is also switched off.

## **EXAMPLE:**

```
class A;  
    rand integer Var1, Var2;  
    static constraint Var1_c { Var1 == 10 ;}  
    constraint Var2_c { Var2 == 10 ;}  
endclass
```

```
program A_p_46;  
    A obj_1 = new;  
    A obj_2 = new;  
    initial  
    begin  
        obj_2.Var1_c.constraint_mode(0);  
        obj_2.Var2_c.constraint_mode(0);  
        repeat(10)  
            begin  
                void'(obj_1.randomize());  
                $display("obj_1.Var1 : %d ,obj_1.Var2 : %d ",obj_1.Var1,obj_1.Var2);  
            end  
    end  
endprogram
```

## **RESULTS:**

```
# obj_1.Var1 : 733126180 ,obj_1.Var2 : 10  
# obj_1.Var1 : -119008195 ,obj_1.Var2 : 10  
# obj_1.Var1 : 342785185 ,obj_1.Var2 : 10  
# obj_1.Var1 : 679818185 ,obj_1.Var2 : 10  
# obj_1.Var1 : -717162992 ,obj_1.Var2 : 10  
# obj_1.Var1 : 664520634 ,obj_1.Var2 : 10  
# obj_1.Var1 : -1944980214 ,obj_1.Var2 : 10  
# obj_1.Var1 : -1350759145 ,obj_1.Var2 : 10  
# obj_1.Var1 : -1374963034 ,obj_1.Var2 : 10  
# obj_1.Var1 : -462078992 ,obj_1.Var2 : 10
```

## **CONSTRAINT EXPRESSION**

A constraint\_expression is any SystemVerilog expression or one of the constraint specific operators(  $\rightarrow$  (Implication) and dist). Functions are allowed to certain limitation. Operators which has side effects are not allowed like  $++$ , $--$ .

## **Set Membership**

A set membership is a list of expressions or a range. This operator searches for the existences of the value in the specified expression or range and returns 1 if it is existing.

**EXAMPLE:**

```
class set_mem;
    rand integer Var;
    constraint range { Var inside {0,1,[50:60],[90:100]}; }
    function void post_randomize();
        $write("%0d",Var);
    endfunction
endclass
program set_mem_p_47;
    set_mem obj=new();
    initial
        repeat(10)
            void'(obj.randomize());
    endprogram
```

**RESULTS:**

50\_57\_60\_93\_100\_94\_90\_1\_54\_100\_

If you want to define a range which is outside the set, use negation.

**EXAMPLE:**

```
class set_mem;
    rand bit [0:2] Var;
    constraint range { !( Var inside {0,1,5,6}); }
    function void post_randomize();
        $write("%0d",Var);
    endfunction
endclass
program set_mem_p_48;
    set_mem obj=new();
    initial
        repeat(10)
            void'(obj.randomize());
    endprogram
```

**RESULTS:**

7\_4\_4\_4\_7\_2\_2\_3\_2\_7\_

Engineers often mistaken that set membership operator is used only in constraint block. It can also be used in other scopes also.

```
class set_mem;
    rand bit [0:2] Var;
endclass
program set_mem_p_48;
    set_mem obj=new();
```

```

integer repet = 0;
initial
begin
    obj.Var = 1;
    repeat(10)
        begin
            void'(obj.randomize());
            while ( obj.Var inside {[1:5]}) 
                begin
                    $display("Var = %0d",obj.Var);
                    break;
                end
            end
        end
    endprogram

```

#### RESULTS:

```

# Var = 4
# Var = 5
# Var = 1
# Var = 1
# Var = 2
# Var = 2

```

NOTE: X and Z are allowed in set membership operator, but not in constraint block, inside {....} is a statement which returns 0,1 or X .

Expressions are allowed in value set of inside operator.

```

rand integer y,x;
constraint c1 {x inside {3, 5,[y:2*y], z};}

```

If an expression in the list is an array then just use the array name in the constraint block.

Elements are traversed by descending into the array until reaching a singular value.

```

int array [$] = '{3,4,5};
if ( ex inside {1, 2, array} )

```

#### Weighted Distribution

There are two types of distribution operators.

The := operator assigns the specified weight to the item or, if the item is a range, to every value in the range.

The :/ operator assigns the specified weight to the item or, if the item is a range, to the range as a whole. If there are n values in the range, the weight of each value is range\_weight / n.

```

Var dist { 10 := 1; 20 := 2 ; 30 := 2 }

```

The probability of Var is equal to 10,20 and 30 is in the ratio of 1,2,2 respectively.

```
Var dist { 10 := 1; 20 := 2 ; [30:32] := 2 }
```

The probability of Var is equal to 10,20,30,31 and 32 is in the ratio of 1,2,2,2,2 respectively.

If you use the := operator each element weight of the range has the assigned weight.

If you want to weight for the whole group, use :/ and the weight is equally distributed for each element in that group.

```
Var dist { 10 := 1; 20 := 2 ; [30:32] :/ 2 }
```

The probability of Var is equal to 10,20,30,31,32 is in the ratio of 1,2,2/3,2/3,2/3 respectively.

To demonstrate the distribution property, here is an example.

**EXAMPLE:**

```
class Dist;  
    rand integer Var;  
    constraint range { Var dist { [0:1] := 50 , [2:7] := 50 }; }  
endclass
```

```
program Dist_p_49;  
    Dist obj;  
    integer count_0, count_1, count_2, count_3, count_4, count_5, count_6, count_7;  
    integer count_0_1 ,count_2_7 ;  
  
initial  
begin  
    obj=new();  
    count_0 = 0;count_1 = 0;count_2 = 0;count_3 = 0;  
    count_4 = 0;count_5 = 0;count_6 = 0;count_7 = 0;  
    count_0_1 = 0;count_2_7 = 0;  
    for(int i=0; i< 10000; i++)  
        if( obj.randomize())  
        begin  
            if( obj.Var == 0) count_0 ++;  
            else if( obj.Var == 1) count_1 ++;  
            else if( obj.Var == 2) count_2 ++;  
            else if( obj.Var == 3) count_3 ++;  
            else if( obj.Var == 4) count_4 ++;  
            else if( obj.Var == 5) count_5 ++;  
            else if( obj.Var == 6) count_6 ++;  
            else if( obj.Var == 7) count_7 ++;  
            if( obj.Var inside {0,1} ) count_0_1 ++;  
            else if( obj.Var inside {[2:7]} ) count_2_7 ++;  
        end
```

```

$display(" count_0 = %0d , count_1 = %0d, count_2 = %0d, count_3 = %0d, count_4 = %0d,
count_5 = %0d, count_6 = %0d, count_7= %0d
",count_0, count_1, count_2, count_3, count_4, count_5, count_6, count_7);
$display(" count_0_1 = %0d ;count_2_7 = %0d ",count_0_1,count_2_7);
$finish;
end
endprogram

```

### RESULTS:

```

# count_0 = 1290 , count_1 = 1244, count_2 = 1286, count_3 = 1265, count_4 = 1230, count_5
= 1243, count_6 = 1189, count_7= 1253
# count_0_1 = 2534 ;count_2_7 = 7466

```

Now change the constraint to

```
constraint range { Var dist { [0:1] :/ 50 , [2:7] :/ 50 }; }
```

### EXAMPLE:

```

class Dist;
rand integer Var;
constraint range { Var dist { [0:1] :/ 50 , [2:7] :/ 50 }; }
endclass

```

```

program Dist_p_50;
Dist obj;
integer count_0, count_1, count_2, count_3, count_4, count_5, count_6, count_7;
integer count_0_1 ,count_2_7 ;
initial
begin
obj=new();
count_0 = 0;count_1 = 0;count_2 = 0;count_3 = 0;
count_4 = 0;count_5 = 0;count_6 = 0;count_7 = 0;
count_0_1 = 0;count_2_7 = 0;

for(int i=0; i< 10000; i++)
if( obj.randomize())
begin
if( obj.Var == 0) count_0++;
else if( obj.Var == 1) count_1++;
else if( obj.Var == 2) count_2++;
else if( obj.Var == 3) count_3++;
else if( obj.Var == 4) count_4++;
else if( obj.Var == 5) count_5++;
else if( obj.Var == 6) count_6++;

```

```

else if( obj.Var == 7) count_7++;
if( obj.Var inside {0,1} ) count_0_1++;
else if( obj.Var inside {[2:7]} ) count_2_7++;
end

$display(" count_0 = %0d , count_1 = %0d, count_2 = %0d, count_3 = %0d, count_4 = %0d,
count_5 = %0d, count_6 = %0d, count_7= %0d
",count_0, count_1, count_2, count_3, count_4, count_5, count_6, count_7);
$display(" count_0_1 = %0d ;count_2_7 = %0d ",count_0_1,count_2_7);
$finish;
end
endprogram

```

### RESULTS:

```

# count_0 = 2496, count_1 = 2508, count_2 = 846, count_3 = 824, count_4 = 833, count_5 =
862, count_6 = 820, count_7= 811
# count_0_1 = 5004 ;count_2_7 = 4996

```

Both the results show, how many times each value occurred.

NOTE: If no weight is specified for items, the default weight is 1. Weight 0 is also allowed.

NOTE: Variable declared as randc are not allowed int dist.

If there are constraints on some expressions that cause the distribution weights on these expressions to be not satisfiable, implementations are only required to satisfy the non dist constraints. Use dist only on a one variable in a set of constraint expression variable.  
In the following example, Even though probability of Var2 is equal to 0 to 1 is in ratio of 50, 50 to satisfy other constraints, the dist is ignored.

### EXAMPLE:

```

class Dist;
  rand integer Var1,Var2;
  constraint dist_c { Var2 dist { [0:1] := 50 , [2:7] := 50 };}
  constraint relation_c { Var1 < Var2; }
  constraint range_c { Var2 inside {2,3,4};}
endclass

program Dist_p_51;
  Dist obj;
  integer count_0, count_1, count_2, count_3, count_4, count_5, count_6, count_7;
  integer count_0_1 ,count_2_7 ;

  initial
  begin

```

```

obj=new();
count_0 = 0;count_1 = 0;count_2 = 0;count_3 = 0;
count_4 = 0;count_5 = 0;count_6 = 0;count_7 = 0;
count_0_1 = 0;count_2_7 = 0;

for(int i=0; i< 10000; i++)
  if( obj.randomize())
    begin
      if( obj.Var2 == 0) count_0++;
      else if( obj.Var2 == 1) count_1++;
      else if( obj.Var2 == 2) count_2++;
      else if( obj.Var2 == 3) count_3++;
      else if( obj.Var2 == 4) count_4++;
      else if( obj.Var2 == 5) count_5++;
      else if( obj.Var2 == 6) count_6++;
      else if( obj.Var2 == 7) count_7++;
      if( obj.Var2 inside {0,1} ) count_0_1++;
      else if( obj.Var2 inside {[2:7]} ) count_2_7++;
    end

```

```

$display(" count_0 = %0d , count_1 = %0d, count_2 = %0d, count_3 = %0d, count_4 = %0d,
count_5 = %0d, count_6 = %0d, count_7= %0d
",count_0, count_1, count_2, count_3, count_4, count_5, count_6, count_7);
$display(" count_0_1 = %0d ;count_2_7 = %0d ",count_0_1,count_2_7);
$finish;
end
endprogram

```

#### RESULTS:

```

# count_0 = 0 , count_1 = 0, count_2 = 3362, count_3 = 3321, count_4 = 3317, count_5 = 0,
count_6 = 0, count_7= 0
# count_0_1 = 0 ;count_2_7 = 10000

```

Calling a new(), resets the distribution algorithm. If new() is called to create an object when ever randomized is called, the dist algorithm restarts and the distribution may not be what is expected.

#### EXAMPLE:

```

class Dist;
  rand integer Var;
  constraint range { Var dist { [0:1] := 50 , [2:7] := 50 };}
endclass

program Dist_p_52;
  Dist obj;

```

```

integer count_0_1 ,count_2_7 ;
initial
begin
    obj=new();
    count_0_1 = 0;count_2_7 = 0;
    for(int i=0; i< 10000; i++)
    begin
        obj = new();
        if( obj.randomize())
            if( obj.Var inside {0,1} ) count_0_1++;
            else if( obj.Var inside {[2:7]} ) count_2_7++;
    end
    $display("count_0_1 : %0d : count_2_7 : %0d ",count_0_1,count_2_7);
end
endprogram

```

#### RESULTS:

# count\_0\_1 : 3478 : count\_2\_7 : 6522

The distribution is not followed if the variable is declared as randc as distribution may require repetition and randc does not allow. In the below example try changing the weights in distribution function, you will get always same results otherwise compilation error.

#### EXAMPLE:

```

class Dist;
    randc byte Var;
    constraint range { Var dist { [0:1] := 50 , [2:7] := 50 };}
endclass

program Dist_p_52;
Dist obj;
integer count_0_1 ,count_2_7 ;
initial
begin
    obj=new();
    count_0_1 = 0;count_2_7 = 0;
    for(int i=0; i< 10000; i++)
    begin
        if( obj.randomize())
            if( obj.Var inside {0,1} ) count_0_1++;
            else if( obj.Var inside {[2:7]} ) count_2_7++;
    end
    $display("count_0_1 : %0d : count_2_7 : %0d ",count_0_1,count_2_7);
end

```

### **endprogram**

To constraint enumerated values using dist, if they are less number, give weights individually. If you want to give weights to enumerated values in groups, then give a continues group. As enumeration is represented in integer, the discontinuation in the declaration may not result properly.

**EXAMPLE:**

```
program enums_53;
typedef enum {V_SMALL,SMALL,MEDIUM,LARGE,V_LARGE} size_e;
class dist_c;
    rand size_e size_d;
    constraint size_dist{size_d dist {[V_SMALL:MEDIUM] :/40,[LARGE:V_LARGE] :/ 60}; }
endclass
```

```
initial begin
    dist_c obj;
    obj=new;
    for (int i=0; i<=10; i++)
        begin
            obj.randomize();
            $display (" size_d = %0s ", obj.size_d);
        end
    end
endprogram
```

### **Implication**

Implication operator can be used to declare conditional relation. The syntax is expression  $\rightarrow$  constraint set. If the expression is true, then the constraint solver should satisfy the constraint set. If the expression is false then the random numbers generated are unconstrained by constraint set. The boolean equivalent of  $a \rightarrow b$  is  $(\neg a \vee b)$ .

```
rand bit a;
rand bit [3:0] b;
constraint c { (a == 0) -> (b == 1); }
```

a is one and b is 4 bit, So there are total of 32 solutions. But due to constraint c (when ever a is 0 b should be 1) 15 solutions are removed. So probability of a = 0 is  $1/(32-15) = 1/17$ . If u observe the following program results count\_0/count\_1 approximately equal to 1/17.

**EXAMPLE:**

```
class impli;
    rand bit a;
    rand bit [3:0] b;
    constraint c { (a == 0) -> (b == 1); }
endclass
```

```

program impli_p_54;
    impli obj;
    integer count_0 ,count_1 ;

initial
begin
    obj=new();
    count_0 = 0;count_1 = 0;
    for(int i=0; i< 10000; i++)
    begin
        if( obj.randomize())
        if( obj.a == 0 ) count_0++;
        else count_1++;
    end
    $display(" count_0 = %0d;count_1 = %0d; ",count_0 ,count_1);
end
endprogram

```

#### RESULTS:

```
# count_0 = 571;count_1 = 9429;
```

#### If..Else

Just like implication, if...else style constraints are bidirectional. Above example applies here too.

#### EXAMPLE:

```

class if_else;
    rand bit a;
    rand bit [3:0] b;
    constraint c { if(a == 0) (b == 1); }
endclass

```

```

program if_else_p_55;
    if_else obj;
    integer count_0 ,count_1 ;

initial
begin
    obj=new();
    count_0 = 0;count_1 = 0;
    for(int i=0; i< 10000; i++)
    begin
        obj = new();
        if( obj.randomize())

```

```

begin
  if( obj.a == 0 ) count_0++;
  else if( obj.a == 1 ) count_1++;
end
end
$display(" count_0 = %0d;count_1 = %0d;",count_0 ,count_1);
end
endprogram

```

### RESULTS:

```
# count_0 = 571;count_1 = 9429;
```

### VARIABLE ORDERING

SystemVerilog constraints provide a mechanism for ordering variables so that some variables can be chosen independently of some variables. The solution space remains the same, but the probability of picking up the solution space changes. The syntax for variable ordering is "solve x before y". The exact meaning of this statement is "choos x before y" as the this statement is to guide the distribution, but not the solution space.

Only rand variables are allowed.

### EXAMPLE:

```

class Var_order_56;
  rand bit a;
  rand bit [3:0] b;
  constraint bidirectional { a -> b == 0; }
endclass

```

The probability of a=1 is  $1/((2^{**}5)-15)=1/17$ , as constraints are bidirectional i.e both the values of a and b are determined together. Constraints will be solved only once, the solver picks the one solution from the possible set of {a,b} which has 17 solutions. To guide the probability of selecting a= 0 to 50% and a = 1 to 50%, use

```
constraint order { solve a before b ; }
```

This guides the solver to give highest priority to a than b while picking the solution from solution space. This is explicit variable ordering. The solver follows the implicit variable ordering also, like randc are solved before rand variables. In dynamic arrays size and elements are solved with two constraints( size constraint and element value constraint) ,array size is solved before element.

### EXAMPLE:

```

class if_57;
  rand bit a;
  rand bit [3:0] b;

```

```

constraint c { if(a == 0) (b == 1); }
constraint order { solve a before b; }
endclass

program if_p;
if_57 obj;
integer count_0 ,count_1 ;

initial
begin
  count_0 = 0;count_1 = 0;
  for(int i=0; i< 10000; i++)
    begin
      obj = new();
      if( obj.randomize())
        if( obj.a == 0 ) count_0++;
        else if( obj.a == 1 ) count_1++;
    end
    $display(" count_0 = %0d;count_1 = %0d;",count_0 ,count_1);
  end
endprogram

```

#### RESULTS:

# count\_0 = 4974;count\_1 = 5026;

Too many explicit variable ordering may lead to circular dependency. The LRM says that "Circular dependencies created by the implicit variable ordering shall result in an error." and "circular dependency is not allowed". But it does not put restriction on what to do if a explicit circular dependency exists. Check with your tool, if explicit Circular dependency is existing, it may report warning,it may fail solver or proceed by just ignoring the order.

#### EXAMPLE:

```

program Cir_Dip_p_58;
class Cir_Dep;
  rand integer a,b,c;
  constraint a_c { solve a before b ;}
  constraint b_c { solve b before c ;}
  constraint c_c { solve c before a ;}
endclass

```

```

Cir_Dip obj=new();
initial
  void'(obj.randomize());

```

**endprogram**

**RESULTS:**

Error: Cird.sv(14): Cyclical dependency between random variables specified by solve/before constraints.

LRM says, if the outcome is same, solver can solve without following the rule. In the following case, x has only one possible assignment (0), so x can be solved for before y. The constraint solver can use this flexibility to speed up the solving process.

**EXAMPLE:**

```
class slove_before;
    rand integer x,y;
    constraint C {x == 0;
        x < y;
        solve y before x; }
endclass
program s_b_59;
    slove_before obj ;
    initial
    begin
        obj = new();
        repeat(5)
            if(obj.randomize())
                $display(" x : %d :: y :%d ",obj.x,obj.y);
            else
                $display("Randomization failed ");
    end
endprogram
```

**RESULTS:**

```
# x :      0 :: y : 2064490291
# x :      0 :: y : 2035140763
# x :      0 :: y : 1279931677
# x :      0 :: y : 2112945927
# x :      0 :: y : 1977312554
```

### Functions

Functions are allowed in constraints. It will be useful in applications like constrain the max packet size based on the bandwidth other parameters. Constraint statements are more error pron. Functions in constraints are called before constraints are solved. The return value of the function is used to solve the constraint and it is treated as state variable. There is an implicit variable ordering when solving these constraints. Function should not be declared as static and should not modify the constraints, for example calling the rand\_mode and constraint\_mode methods.

#### **EXAMPLE:**

```
class B_61;
  rand int x, y;
  constraint C { x <= F(y); }
  constraint D { y inside { 2, 4, 8 } ; }
endclass
```

Random variables used as function arguments shall establish an implicit variable ordering or priority. Constraints that include only variables with higher priority are solved before other lower priority constraints. Random variables solved as part of a higher priority set of constraints, become state variables to the remaining set of constraints. In above example y is solved before x. If y is not rand variable, then F(y) is equivalent to calling it in the pre\_randomize method.

#### **Iterative Constraints**

Iterative constraints allow Constraining individual elements of fixed-size, dynamic, associative arrays or queue. foreach construct specifies iteration over the each elements of array.

#### **EXAMPLE:**

```
class Eth_pkt_60;
  rand byte Payload[];
  constraint size_c { Payload.size() inside {[10:1500]}; }
  constraint element_c { foreach ( Payload[i] ) Payload[ i ] inside {[50:100]}; }
  function void post_randomize();
    foreach(Payload[i])
      $display( " Payload[ %0d ] :%d ",i,Payload[ i ] );
  endfunction
endclass
```

```
program iterative_60;
  Eth_pkt_60 obj;
  initial
  begin
    obj = new();
    if(obj.randomize())
      $display(" RANDOMIZATION DONE ");
  end
endprogram
```

#### **RESULTS:**

```
# Payload[ 0 ] : 87
# Payload[ 1 ] : 52
# Payload[ 2 ] : 70
# Payload[ 3 ] : 76
```

```

# Payload[ 4 ] : 71
# Payload[ 5 ] : 63
# Payload[ 6 ] : 62
# Payload[ 7 ] : 63
# Payload[ 8 ] : 66
# Payload[ 9 ] : 85
# Payload[ 10 ] : 95
# Payload[ 11 ] : 57
# Payload[ 12 ] : 78

```

**NOTE:** Remember **while** randomizing each element of array also array should be declared as random variable otherwise randomization wont happen.

### **CONSTRAINT SOLVER SPEED**

IEEE SystemVerilog 2005 LRM does not specify the constraint solver algorithm. So different vendors use different constraint solver algorithms. Some vendors may get the solution quickly, some may not.

All the points discussed in this are not specific to a vendor. Just check your tool with these examples to know how ur vendor constraint solver works.

As it is handy to type int and integer, we always declare some data types like size of dynamic arrays as integer and constraint it to required range.

This will create  $2^{32}$  values and solver has to choose random number and solve the constraint which will hit only few times.

Probability of a random value to be in solution space is less than appearing it outside the solution space. To speed up ur solver, use the proper data types.

In the following example i showed the results of Questasim tool.

integer is 32 bit datatype.

shortint is 16 bit datatype.

byte is 8 bit datatype.

**EXAMPLE:**

```

program inte_p_62;
  class inte;
    rand integer Var;
    constraint randge_c { Var inside {[0:10]};}
  endclass
  inte obj=new();
  initial
  begin
    repeat(10000)
      void'(obj.randomize());
    $finish(2);
  
```

```
end  
endprogram
```

#### RESULT:

```
# ** Note: Data structure takes 3407960 bytes of memory  
#      Process time 2.78 seconds
```

#### EXAMPLE:

```
program inte_p_62;  
  class inte;  
    rand bit [4:0] Var;  
    constraint randge_c { Var inside {[0:10]};}  
  endclass  
  inte obj=new();  
  initial  
  begin  
    repeat(10000)  
      void'(obj.randomize());  
      $finish(2);  
    end  
  endprogram
```

#### RESULT:

```
# ** Note: Data structure takes 3407960 bytes of memory  
#      Process time 1.95 seconds
```

To generate a random value which is one of the  $2^{**0}, 2^{**1}, 2^{**2}, \dots, 2^{**31}$ .

If data type is integer and randomizing it will consume lot of simulation time.

```
constraint C { Var inside {2**0,2**1,2**2,...,2**31};}
```

Write a function to check only one bit is high.

```
constraint C { $countones(array) == 1; }  
constraint C { Var == (1 << index); }
```

Instead declare an 5 bit variable index, randomize it and set the index bit to 1 in post randomize.

#### EXAMPLE:

```
class expo_cons_64;  
  rand bit [0:4] index;  
  integer array; // No need to randomize array  
  function void post_randomize;  
    array = 'b0;  
    array[index]=1'b1;  
  endfunction  
endclass
```

Adding smaller constraints in to a single block, speeds up simulation for some simulators. Check these examples with your simulator. Some Solver may need more iterations to get the valid solution. Split the constraints as small as possible. With this controllability on constraint blocks is also improved.

#### EXAMPLE:1

```
class constr;
  rand int a,b,c,d;
  constraint C { (a == 10)&&( b < 20 ) && (c > 30) && (d < 40) ;}
endclass

program constr_p_65;
  constr obj;
  initial
  begin
    obj=new();
    repeat(10000)
      void'(obj.randomize());
    $finish(2);
  end
endprogram
```

#### RESULT:

```
# ** Note: Data structure takes 3276888 bytes of memory
#       Process time 9.44 seconds
```

#### EXAMPLE:2

```
class constr;
  rand int a,b,c,d;
  constraint Ac {(a == 10);}
  constraint Bc {(b < 20);}
  constraint Cc {(c > 30);}
  constraint Dc {(d < 40);}
endclass

program constr_p_66;
  constr obj;
  initial
  begin
    obj=new();
    repeat(10000)
      void'(obj.randomize());
    $finish(2);
  end
endprogram
```

**RESULT:**

```
# ** Note: Data structure takes 3407960 bytes of memory
#       Process time 9.27 seconds
```

**EXAMPLE:3**

```
class constr_67;
  rand int a,b,c,d;
  constraint Ac { (a == 10) ; ( b < 20 ) ; (c > 30) ; (d < 40) ;}
endclass
```

**program** constr\_p\_67;

```
constr_p_67 obj;
initial
begin
  obj=new();
  repeat(10000)
    void'(obj.randomize());
  $finish(2);
end
endprogram
```

**RESULT:**

```
# ** Note: Data structure takes 3407960 bytes of memory
#       Process time 9.24 seconds
```

Run all the above three examples with your simulator and check how your simulation speed varies.

When iterative constraints are used on arrays, each element has a constraint on it. If the constraints are simple enough to implement with out using constraint block, simulation time may be saved. In the following example, there are two constraints blocks, one for size other for each element. In reality there may be more constraint blocks.

**EXAMPLE:**

```
class Eth_pkt;
  rand byte Payload[];
  constraint size_c { Payload.size() inside {[46:1500]}; }
  constraint element_c { foreach ( Payload[ i ] ) Payload[ i ] inside {[50:100]}; }
endclass
```

**program** iterative\_68;

```
Eth_pkt obj;
initial
begin
  obj = new();
```

```

for(int i=0;i< 10000;i++)
begin
  if(obj.randomize())
    $display(" RANDOMIZATION DONE ");
end
$finish(2);
end
endprogram
RESULT:
# ** Note: Data structure takes 3407960 bytes of memory
#      Process time 705.51 seconds
The above logic can implemented using post_randomize. Check how these two example with ur
vendor tool and look at the simulation speed. You may find difference.
EXAMPLE:
class Eth_pkt;
  rand integer length;
  byte Payload[];
  constraint size_c { length inside {[46:1500]}; }
  function void post_randomize;
    Payload = new[length];
    for(int i=0;i< length;i++)
      Payload[ i ] = 50 + $urandom % 51 ;
  endfunction
endclass

program iterative_69;
  Eth_pkt obj;
  initial
  begin
    obj = new();
    for(int i=0;i< 10000;i++)
      begin
        if(obj.randomize())
          $display(" RANDOMIZATION DONE ");
      end
      $finish(2);
    end
endprogram
# ** Note: Data structure takes 3539032 bytes of memory
#      Process time 3.92 seconds

```

## **RANDCASE**

randcase is a case statement that randomly selects one of its branches. The randcase item expressions are non-negative integral values that constitute the branch weights. An item weight divided by the sum of all weights gives the probability of taking that branch. Randcase can be used in class and modules. The randcase weights can be arbitrary expressions, not just constants.

**EXAMPLE:**

**randcase**

3 : x = 1;

1 : x = 2;

4 : x = 3;

**endcase**

The sum of all weights is 8; therefore, the probability of taking the first branch is 0.375, the probability of taking the second is 0.125, and the probability of taking the third is 0.5. If a branch specifies a zero weight, then that branch is not taken. The sum of all weights (SUM) is computed (negative values contribute a zero weight). If SUM is zero or exceeds (2\*32-1), no branch is taken. Each call to randcae statement will return a random number in the range from 0 to SUM. \$urandom\_range(0,SUM) is used to generate a random number. As the random numbers are generated using \$urandom are thread stable, randcase also exhibit random stability.

**EXAMPLE:**

```
module rand_case_70;
    integer x;
    integer cnt_1,cnt_2,cnt_3;
    initial
    begin
        cnt_1 = 0;cnt_2=0;cnt_3 = 0;
        repeat(100000)
            begin
                randcase
                    3 : x = 1;
                    1 : x = 2;
                    4 : x = 3;
                endcase
                if(x == 1)
                    cnt_1++;
                else if(x == 2)
                    cnt_2++;
                else if(x ==3)
                    cnt_3++;
            end
            $display("count_1 =%0d count_2 =%0d count_3 =%0d ",cnt_1,cnt_2,cnt_3);
    end
```

```

$display("Probability of count_1 =%0f count_2 =%0f count_3 =%0f
",(cnt_1/100000.0),(cnt_2/100000.0),(cnt_3/100000.0));
end
endmodule

```

#### **RESULTS:**

```

# count_1 =37578 count_2 =12643 count_3 =49779
# Probability of count_1 =0.375780 count_2 =0.126430 count_3 =0.497790

```

### **RANDSEQUENCE**

The random sequence generator is useful for randomly generating sequences of stimulus. For example, to verify a temporal scenario, a sequence of packets are needed. By randomizing a packet, it will generate most unlikely scenarios which are not interested. These type of sequence of scenarios can be generated using randsequence. A randsequence grammar is composed of one or more productions. Production items are further classified into terminals and nonterminals. A terminal is an indivisible item that needs no further definition than its associated code block.

#### **EXAMPLE:**

```

module rs();
initial
begin
repeat(5)
begin
randsequence( main )
    main : one two three ;
    one : {$write("one");};
    two : {$write(" two");};
    three: {$display(" three");};
endsequence
end
end
endmodule

```

#### **RESULTS:**

```

one two three

```

The production main is defined in terms of three nonterminals: one, two and three. Productions one,two and three are terminals. When the main is chosen, it will select the sequence one, two and three in order.

### **Random Productions:**

A single production can contain multiple production lists separated by the | symbol. Production lists separated by a | imply a set of choices, which the generator will make at random.

#### **EXAMPLE:**

```
module rs();
initial
repeat(8)
randsequence( main )
main : one | two | three ;
one : {$display("one");};
two : {$display("two");};
three: {$display("three");};
endsequence
```

```
endmodule
```

#### **RESULTS:**

```
# three
# three
# three
# three
# one
# three
# two
# two
```

Results show that one, two and three are selected randomly.

### **Random Production Weights :**

The probability that a production list is generated can be changed by assigning weights to production lists. The probability that a particular production list is generated is proportional to its specified weight. The := operator assigns the weight specified by the weight\_specification to its production list. A weight\_specification must evaluate to an integral non-negative value. A weight is only meaningful when assigned to alternative productions, that is, production list separated by a |. Weight expressions are evaluated when their enclosing production is selected, thus allowing weights to change dynamically.

#### **EXAMPLE:**

```
module rs();
integer one_1,two_2,three_3;
initial
begin
one_1 = 0;
two_2 = 0;
three_3 = 0;
```

```

repeat(6000)
  randsequence( main )
    main : one := 1| two := 2| three := 3;
    one : {one_1++;};
    two : {two_2++;};
    three: {three_3++;};
  endsequence
  $display(" one %0d two %0d three %0d",one_1,two_2,three_3);
end
endmodule

```

#### RESULTS:

# one 1011 two 2005 three 2984

#### If..Else

A production can be made conditional by means of an if..else production statement. The expression can be any expression that evaluates to a boolean value. If the expression evaluates to true, the production following the expression is generated, otherwise the production following the optional else statement is generated.

#### EXAMPLE:

```

module rs();
  integer one_1,two_2,three_3;
  reg on;
  initial
  begin
    on = 0;
    one_1 = 0;
    two_2 = 0;
    three_3 = 0;
    repeat(2500)
      randsequence( main )
        main : one three;
        one : if(on) one_1++; else two_2++;
        three: {three_3++;};
      endsequence
      $display(" one %0d two %0d three %0d",one_1,two_2,three_3);
    end
endmodule

```

#### RESULTS:

# one 0 two 2500 three 2500

## Case

A production can be selected from a set of alternatives using a case production statement. The case expression is evaluated, and its value is compared against the value of each case-item expression, which are evaluated and compared in the order in which they are given. The production associated with the first case-item expression that matches the case expression is generated. If no matching case-item expression is found then the production associated with the optional default item is generated, or nothing if there no default item. Case-item expressions separated by commas allow multiple expressions to share the production.

**EXAMPLE:**

```
module rs();
integer one_1,two_2,three_3;
initial
begin
    one_1 = 0;
    two_2 = 0;
    three_3 = 0;
    for(int i = 0 ;i < 6 ;i++)
begin
    randsequence( main )
    main : case(i%3)
        0 : one;
        1 : two;
        default: def;
    endcase;
    one : {$display("one");}
    two : {$display("two");}
    def : {$display("default");}
endsequence
end
end
endmodule
```

**RESULTS:**

```
one
two
default
one
two
default
```

### **Repeat Production Statements :**

The repeat production statement is used to iterate a production over a specified number of times. The repeat production statement itself cannot be terminated prematurely. A break statement will terminate the entire randsequence block

PUSH\_OPER : repeat( \$urandom\_range( 2, 6 ) ) PUSH ;

Interleaving productions-rand join :

**EXAMPLE:**

```
module rs();
integer one_1,two_2,three_3;
initial
begin
one_1 = 0;
two_2 = 0;
three_3 = 0;
repeat(6000)
randsequence( main )
main : one | repeat(2) two | repeat (3) three ;
one : one_1++;
two : two_2++;
three: three_3++;
endsequence
$display(" one %d two %d three %d",one_1,two_2,three_3);
end
endmodule
```

### **RESULTS:**

one 989 two 2101 three 2810

### **Rand Join**

The rand join production control is used to randomly interleave two or more production sequences while maintaining the relative order of each sequence.

**EXAMPLE:**

```
module rs();
initial
for(int i = 0;i < 24;i++) begin
randsequence( main )
main : rand join S1 S2 ;
S1 : A B ;
S2 : C D ;
A : $write("A");

```

```

B : $write("B");
C : $write("C");
D : $write("D");
endsequence
if(i%4 == 3 )
    $display("");
end
endmodule

```

### RESULTS:

```

A B C D
A C B D
A C D B
C D A B
C A B D
C A D B

```

Note that B always comes after A and D comes after C. The optional expression following the rand join keywords must be a real number in the range 0.0 to 1.0. The value of this expression represents the degree to which the length of the sequences to be interleaved affects the probability of selecting a sequence. A sequences length is the number of productions not yet interleaved at a given time. If the expression is 0.0, the shortest sequences are given higher priority. If the expression is 1.0, the longest sequences are given priority.

### EXAMPLE:

```

module rs();
    initial
        for(int i = 0;i < 24;i++)
            begin
                randsequence( main )
                    main : rand join (0.0) S1 S2 ;
                    S1 : A B ;
                    S2 : C D ;
                    A : $write("A");
                    B : $write("B");
                    C : $write("C");
                    D : $write("D");
                endsequence
                if(i%4 == 3 )
                    $display("");
            end
    endmodule

```

## RESULTS:

A B C D  
C D A B  
A C B D  
A C D B  
C A B D  
C A D B

## EXAMPLE:

```
module rs();
    initial
    for(int i = 0;i < 24;i++) begin
        randsequence( main )
            main : rand join (1.0) S1 S2 ;
            S1 : A B ;
            S2 : C D ;
            A : $write("A");
            B : $write("B");
            C : $write("C");
            D : $write("D");
    endsequence
    if(i%4 == 3 )
        $display("");
    end
endmodule
```

## RESULTS:

A C B D  
A C D B  
C A D B  
C A D B  
A B C D  
C D A B  
C A B D

## Break

The break statement terminates the sequence generation. When a break statement is executed from within a production code block, it forces a jump out the randsequence block.

## EXAMPLE:

```
randsequence()
    WRITE : SETUP DATA ;
    SETUP : { if( fifo_length >= max_length ) break; } COMMAND ;
    DATA : ...
```

### **endsequence**

When the example above executes the break statement within the SETUP production, the COMMAND production is not generated, and execution continues on the line labeled next\_statement.

### **Return**

The return statement aborts the generation of the current production. When a return statement is executed from within a production code block, the current production is aborted. Sequence generation continues with the next production following the aborted production.

#### **EXAMPLE:**

##### **randsequence()**

```
TOP : P1 P2 ;  
P1 : A B C ;  
P2 : A { if( flag == 1 ) return; } B C ;  
A : { $display( A ); } ;  
B : { if( flag == 2 ) return; $display( B ); } ;  
C : { $display( C ); } ;
```

##### **endsequence**

Depending on the value of variable flag, the example above displays the following:

flag == 0 ==> A B C A B C

flag == 1 ==> A B C A

flag == 2 ==> A C A C

When flag == 1, production P2 is aborted in the middle, after generating A. When flag == 2, production B is aborted twice (once as part of P1 and once as part of P2), but each time, generation continues with the next production, C.

### **Value Passing Between Productions**

Data can be passed down to a production about to be generated, and generated productions can return data to the non-terminals that triggered their generation. Passing data to a production is similar to a task call, and uses the same syntax. Returning data from a production requires that a type be declared for the production, which uses the same syntax as a variable declaration. Productions that accept data include a formal argument list. The syntax for declaring the arguments to a production is similar to a task prototype; the syntax for passing data to the production is the same as a task call.

#### **EXAMPLE:**

##### **randsequence( main )**

```
main : first second gen ;  
first : add | dec ;  
second : pop | push ;  
add : gen("add") ;  
dec : gen("dec") ;
```

```

pop : gen("pop") ;
push : gen("push") ;
gen( string s = "done" ) : { $display( s ); } ;
endsequence

```

## **RANDOM STABILITY**

In verilog,if the source code does not change,with the same seed,the simulator produces the same random stimulus on any machine or any operating system.Verilog has only one Random number generator.Random stimulus is generated using \$random(seed) where the seed is input to the RNG.\$random will always return the same value for same seed.

### **EXAMPLE:**

```

module seed_74();
initial
repeat(5)
    $display("random stimulus is %d",$random(Seed));
endmodule

```

While debugging to produce the same simulation, we should make sure that calls to RNG is not disturbed. In Verilog if source code changes it is very unlikely that same stimulus is produced with the same seed and we miss the bug.

In SystemVerilog seeding will be done hierachily. Every module instance, interface instance, program instance and package has initialization RNG. Every thread and object has independent RNG . When ever dynamic thread is created its RNG is initialized with the next random value from its parent thread. RNG initialization and RNG generation are different process.During RNG initialization,only seed is set to RNG. When ever static thread is created its RNG is initialized with the next random value from the initialization RNG of module instance, interface instance, program interface or package containing thread declaration. RNG initialization and RNG generation are different process. During RNG initialization,only seed is set to RNG.

The random number generator is deterministic. Each time the program executes, it cycles through the same random sequence. This sequence can be made nondeterministic by seeding the \$urandom function with an extrinsic random variable, such as the time of day.

SystemVerilog system functions \$urandom and \$urandom\_range are thread stable. Calls to RNG using these system function, uses the RNG of that thread. So next time while using \$random in SystemVerilog,think twice.

NOTE: The same stimulus sequence can not be produced on different simulators as the LRM does not restrict the vendors to implement specific constraint solver. Verilog LRM specifies the RNG algorithm for \$random ,so the same stimulus can be produced on different simulators(not always as I discussed in one of the above topic). Even if the SystemVerilog LRM specifies RNG algorithm ,the same sequence cannot be produced on different vendors because of the following are few reasons :

-> LRM doesnot restrict the constraint solver algorithm.

-> Order of threads creation.

-> Order of thread execution.

**EXAMPLE:**

```
class Ran_Stb_1;
  rand bit [2:0] Var;
endclass
program Ran_Stb_p_75;
  Ran_Stb_1 obj_1 = new();
  initial
    repeat(10)
      begin
        void'(obj_1.randomize());
        $display(" Ran_Stb_1.Var : %0d ",obj_1.Var);
      end
endprogram
```

**RESULTS:**

```
# Ran_Stb_1.Var : 4
# Ran_Stb_1.Var : 5
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 0
# Ran_Stb_1.Var : 2
# Ran_Stb_1.Var : 2
# Ran_Stb_1.Var : 7
# Ran_Stb_1.Var : 6
# Ran_Stb_1.Var : 0
```

Stimulus generated in a thread or object is indepeded of other stimulus. So changes in the source code will not effect threads or objects. This is Valid as long as the order of the threads is not distrubed. If a new object is created, make sure that that they are added at the end.

**EXAMPLE:**

```
class Ran_Stb_1;
  rand bit [2:0] Var;
endclass
class Ran_Stb_2;
  rand bit [2:0] Var;
endclass
```

```
program Ran_Stb_p_76;
  Ran_Stb_1 obj_1 = new();
```

```

Ran_Stb_2 obj_2 = new();
initial
repeat(10)
begin
void'(obj_1.randomize());
$display(" Ran_Stb_1.Var : %0d ",obj_1.Var);
end
endprogram

```

New object obj\_2 is added after all the objects. Look at the simulation results, they are same as the above.

#### RESULTS:

```

# Ran_Stb_1.Var : 4
# Ran_Stb_1.Var : 5
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 0
# Ran_Stb_1.Var : 2
# Ran_Stb_1.Var : 2
# Ran_Stb_1.Var : 7
# Ran_Stb_1.Var : 6
# Ran_Stb_1.Var : 0

```

If a new thread is added, make sure that it is added after all the threads.

#### EXAMPLE:

```

class Ran_Stb_1;
rand bit [2:0] Var;
endclass
class Ran_Stb_2;
rand bit [2:0] Var;
endclass
program Ran_Stb_p_77;
Ran_Stb_1 obj_1 = new();
Ran_Stb_2 obj_2 = new();
initial
begin
repeat(5)
begin
void'(obj_1.randomize());
$display(" Ran_Stb_1.Var : %0d ",obj_1.Var);
end
repeat(5)

```

```

begin
  void'(obj_2.randomize());
  $display(" Ran_Stb_2.Var : %0d ",obj_2.Var);
end
end
endprogram

```

The results show clearly that Random values in obj\_1 are same as previous program simulation.

#### **RESULTS:**

```

# Ran_Stb_1.Var : 4
# Ran_Stb_1.Var : 5
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 0
# Ran_Stb_2.Var : 3
# Ran_Stb_2.Var : 3
# Ran_Stb_2.Var : 6
# Ran_Stb_2.Var : 1
# Ran_Stb_2.Var : 1

```

Order of the randomize call to different objects doesnot effect the RNG generation.

#### **EXAMPLE:**

```

class Ran_Stb_1;
  rand bit [2:0] Var;
endclass
class Ran_Stb_2;
  rand bit [2:0] Var;
endclass
program Ran_Stb_p_78;
  Ran_Stb_1 obj_1 = new();
  Ran_Stb_2 obj_2 = new();
initial
begin
  repeat(5)
    begin
      void'(obj_2.randomize());
      $display(" Ran_Stb_2.Var : %0d ",obj_2.Var);
    end
    repeat(5)
      begin
        void'(obj_1.randomize());
        $display(" Ran_Stb_1.Var : %0d ",obj_1.Var);
      end
    end
  end

```

```
end  
end  
endprogram
```

#### RESULTS:

```
# Ran_Stb_2.Var : 3  
# Ran_Stb_2.Var : 3  
# Ran_Stb_2.Var : 6  
# Ran_Stb_2.Var : 1  
# Ran_Stb_2.Var : 1  
# Ran_Stb_1.Var : 4  
# Ran_Stb_1.Var : 5  
# Ran_Stb_1.Var : 1  
# Ran_Stb_1.Var : 1  
# Ran_Stb_1.Var : 0
```

Adding a constraint in one class, will only effect the stimuli of that object only.

#### EXAMPLE:

```
class Ran_Stb_1;  
    rand bit [2:0] Var;  
    constraint C { Var < 4 ;}  
endclass  
class Ran_Stb_2;  
    rand bit [2:0] Var;  
endclass  
program Ran_Stb_p_79;  
    Ran_Stb_1 obj_1 = new();  
    Ran_Stb_2 obj_2 = new();  
    initial  
        repeat(5)  
            begin  
                void'(obj_1.randomize());  
                void'(obj_2.randomize());  
                $display(" Ran_Stb_1.Var : %0d :: Ran_Stb_2.Var : %0d ",obj_1.Var,obj_2.Var);  
            end  
endprogram
```

#### RESULTS:

```
# Ran_Stb_1.Var : 0 :: Ran_Stb_2.Var : 3  
# Ran_Stb_1.Var : 1 :: Ran_Stb_2.Var : 3  
# Ran_Stb_1.Var : 1 :: Ran_Stb_2.Var : 6  
# Ran_Stb_1.Var : 1 :: Ran_Stb_2.Var : 1  
# Ran_Stb_1.Var : 0 :: Ran_Stb_2.Var : 1
```

## Srandom

When an object or thread is created, its RNG is seeded using the next value from the RNG of the thread that creates the object. This process is called hierarchical object seeding. Sometimes it is desirable to manually seed an objects RNG using the srandom() method. This can be done either in a class method or external to the class definition. Example to demonstrate seeding in objects.

### **EXAMPLE:**

```
class Rand_seed;
    rand integer Var;
    function new (int seed);
        srandom(seed);
        $display(" SEED is initialised to %0d ",seed);
    endfunction
    function void post_randomize();
        $display(": %0d :",Var);
    endfunction
endclass
program Rand_seed_p_80;
    Rand_seed rs;
    initial
    begin
        rs = new(20);
        repeat(5)
            void'(rs.randomize());
        rs = new(1);
        repeat(5)
            void'(rs.randomize());
        rs = new(20);
        repeat(5)
            void'(rs.randomize());
    end
endprogram
```

### **RESULTS:**

```
# SEED is initialised to 20
# : 1238041889 :
# : 1426811443 :
# : 220507023 :
# : -388868248 :
# : -1494908082 :
# SEED is initialised to 1
# : 1910312675 :
```

```
# : 632781593 :  
# : -453486143 :  
# : 671009059 :  
# : -967095385 :  
# SEED is initialised to 20  
# : 1238041889 :  
# : 1426811443 :  
# : 220507023 :  
# : -388868248 :  
# : -1494908082 :
```

Simulation results show that same sequence is repeated when the same seed is used for initialization.

Example to demonstrate seeding a thread.

**EXAMPLE:**

```
integer x, y, z;  
fork //set a seed at the start of a thread  
begin process::$random(100); x = $urandom; end  
//set a seed during a thread  
begin y = $urandom; process::$random(200); end  
// draw 2 values from the thread RNG  
begin z = $urandom + $urandom ; end  
join
```

The above program fragment illustrates several properties:

Thread locality: The values returned for x, y, and z are independent of the order of thread execution. This is an important property because it allows development of subsystems that are independent, controllable, and predictable.

Hierarchical seeding: When a thread is created, its random state is initialized using the next random value from the parent thread as a seed. The three forked threads are all seeded from the parent thread.

IEEE 2005 SystemVerilog LRM does not specify whether scope randomization function is random stable or not.

From LRM

### 13.13 Random stability

The RNG is localized to threads and objects. Because the sequence of random values returned by a thread or object is independent of the RNG in other threads or objects, this property is called random stability.

Random stability applies to the following:

- ⌚ The system randomization calls, \$urandom() and \$urandom\_range()
- ⌚ The object and process random seeding method, srandom()

## ⌚ The object randomization method, randomize()

In above its mentioned that "object randomization method, randomize()". There is nothing mentioned about std::randomize() method random stability.

## **ARRAY RANDOMIZATION**

Most application require to randomize elements of array. Arrays are used to model payload, port connections etc.

SystemVerilog has Fixed Arrays, Dynamic arrays, queues and Associative arrays.

If an array is constrained by both size constraints and iterative constraints for constraining every element of array. The size constraints are solved first, and the iterative constraints next. In the example, size\_c is solved first before element\_c. As constraint element\_c cannot be solved without knowing the size. So there is implicit ordering while solving this type of constraints.

This is the only approach for applying constraints to individual elements of arrays, with this approach, performance is degraded. If Payload.size() = 1000, then 1000 constraints are created by foreach. It takes much time for the solver to solve all the constraints. Another simple approach is using a task to randomize each element. This approach is 5X times faster than foreach.

### **EXAMPLE:**

```
class Eth_pkt_82;
    rand byte Payload[];
    constraint size_c { Payload.size() inside {[46:1500]}; }
    task randomize_FOREACH;
        foreach (Payload[i])
            Payload[i] = 50 + $urandom % 50; // try with randomize( Payload[i] ) with ....
    endtask
endclass
program iterative;
    Eth_pkt_82 obj;
    initial
    begin
        obj = new();
        for(int i=0;i< 10000;i++)
            begin
                void'(obj.randomize());
                obj.randomize_FOREACH();
            end
    end
endprogram
```

In applications where some constraints are dependent on the array elements, the following may help. Call the randomization function in constraints, define proper variable ordering.

### EXAMPLE:

```
class Eth_pkt;
    rand byte Payload[];
    rand byte size;
    bit dummy = 1;
    constraint dummy_c { dummy == randomize_size();}
    constraint size_c { size inside {[10:100]};}
    constraint order_c1 {solve size before dummy;}
    constraint order_c2 {solve dummy before Payload;}
    function bit randomize_size();
        Payload = new[size];
        randomize_size = 1;
        return randomize_size;
    endfunction
    endclass
program iterative_88;
    Eth_pkt obj;
    initial
    begin
        obj = new();
        void'(obj.randomize());
        for(int i=0;i< 10;i++)
            begin
                void'(obj.randomize());
                $display(" Eth_pkt.size : %d :: Payload.size() : %d ",obj.size,obj.Payload.size());
                for(int j=0;j<obj.Payload.size();j++)
                    $write("%4d ",obj.Payload[j]);
                $display("");
            end
    end
endprogram
```

In the above example, constraint order\_c1 { solve size before dummy;} makes sure that size is randomized before function ranomize\_size() is called.

If constraints are dependent on the sum of the elements of the dynamic array. The interesting elements are the new random variables which are created by current randomization call. If you are not using any techinique to get the right size after randomization sum() returns the sum of all the array elements .

### EXAMPLE:

```
class dynamic_array_89;
    rand byte size;
```

```

rand byte data[];
constraint size_c { data.size() == size; size >= 0; }
constraint sum_c { data.sum() < 1000; }
endclass

```

Do manually in a function and use it or just use { data.sum() with (item.index < size) }  
 There is one more bug in the above code. The sum() method returns a single value of the same type as the array element type. So the sum returns only 8 bits in this case. So a.sum() is always less than 255 which is less than 1000 and always the constraint is satisfied which is not what is expected.

**EXAMPLE:**

```

program summ;
  dynamic_array obj = new();
  integer sum;
initial
  begin
    sum = 0;
    void'(obj.randomize());
    for(int i=0;i< obj.size ;i++)
      sum= sum+obj.data[i];
    $display(" Sum is %d ",sum);
  end
endprogram

```

Using y.sum with (item + 32'b0) will result in a 32 bit proper sum.

**EXAMPLE:**

```

class dynamic_array;
  rand integer size;
  rand reg [7:0] data[];
  constraint sum_c { data.sum() == data[0];}
  constraint size_c { data.size() == size; size >1000 ;size < 2000; }
endclass

program summ;
  dynamic_array obj = new();
  integer sum;
initial
  repeat(10)
    begin
      sum = 0;
      void'(obj.randomize());
      for(int i=0;i< obj.size ;i++)
        begin

```

```

sum= sum+obj.data[i];
end
$display(" Sum is %d obj.sum() %d",obj.data[0],obj.data.sum());
end
endprogram

```

Randomization donot create the objects. So when a array of objects is randomized, all the objects are pointing to null and randomization can not be done on null objets.new() has to be called to creat objects then only randomize can be done on it.So creat array of objects before calling the randomize.

**EXAMPLE:**

```

class cls;
  rand integer Var;
endclass

class arr_obj;
  rand cls objcls [0:2];
endclass

program arr_obj_p_91;
  arr_obj obj = new() ;
  int i;
  initial
  begin
    if(obj.randomize())
    begin
      $display(" Randomization is done ");
      for(i=0;i<3;i++)
        if(obj.objcls[i] == null)
          $display( " obj.objcls == null ");
        else
          $display(" obj.objcls.Var : %d ", obj.objcls[i].Var );
    end
    else
      $display(" Randomization failed ");
  end
endprogram

```

**RESULTS:**

```

# Randomization is done
# obj.objcls == null
# obj.objcls == null
# obj.objcls == null

```

In the following, objects are created during the creation of arr\_obj. This can also be done in pre\_randomize. When dynamic arrays of objects are created, similar approach has to be taken and size of the dynamic array has to be decided before the new() is called, which makes no sense using the dynamic array of objects.

**EXAMPLE:**

```
class cls;
  rand integer Var;
endclass
class arr_obj;
  rand cls objcls [0:2];
function new();
  foreach(objcls[i])
    objcls[i]=new();
endfunction
endclass
program arr_obj_p_91;
  arr_obj obj = new();
  int i;
  initial
    begin
      if(obj.randomize())
        begin
          $display(" Randomization is done ");
          for(i=0;i<3;i++)
            if(obj.objcls[i] == null )
              $display( " obj.objcls == null ");
            else
              $display(" obj.objcls.Var : %d ", obj.objcls[i].Var );
        end
      else
        $display(" Randomization failed ");
    end
endprogram
```

**RESULTS:**

Randomization is done  
obj.objcls.Var : 733126180  
obj.objcls.Var : -119008195  
obj.objcls.Var : 342785185

To create queue of objects, first length of the queue has to be randomized. Then number of objects equal to length of queue. Delete the old elements in the queue. Then push each object new objects in to the queue. Lastly randomize each object.

**EXAMPLE:**

```
class cls;
    rand integer Var;
endclass

class q_cls;
    rand cls obj[$];
    rand bit [2:0] length;
function void pre_randomize();
    length = $urandom % 20; // Compute the length of queue.
    obj = {} ;           // Delete all the elements in the queue or .delet can be used
    repeat(length)
        begin
            cls obj_loc;
            obj_loc = new(); // Create new object.
            obj.Push_back(obj_loc); // Insert it into queue.
        end
    endfunction
endclass

program q_obj_p_93;
q_cls obj = new();
initial
begin
    if(obj.randomize())
begin
    $display( "Randomization done");
    $write( " Length of q : %0d :::",obj.length);
    for(int i ;i< obj.length;i++)
begin
    cls obj_temp;
    obj_temp = obj.obj.Pop_front();
    $write(" : %0d : ",obj_temp.Var);
end
end
else
    $display( "Randomization failed");
end
endprogram
```

## RESULT:

Randomization done

Length of q : 6 :::: 1474208060 :: -1098913923 :: 816460770 :: 41501707 :: -1179418145 :: -212817600 :

Some application like linked list needs to generate an array of random values which are unique. foreach provides the solution, but the simplest and the best solution is to assign all the elements in array with its index and use shuffle inside a task.shuffle() randomizes the order of the elements in the array. Constraint solver is not used here, so the performance is better.

## EXAMPLE:

```
class List;
    integer Pointer[5];
    task randomize_unique;
        foreach ( Pointer[ i ] )
            Pointer[ i ] = i;
        Pointer.shuffle();
    endtask
endclass
program Unique_rand_94;
List obj;
initial
begin
    obj = new();
    obj.randomize_unique();
    for(int i=0;i< 5;i++)
        begin
            $display(" Pointer[%0d] = %d ",i,obj.Pointer[i]);
        end
    end
endprogram
```

## RESULT:

```
Pointer[0] =      2
Pointer[1] =      1
Pointer[2] =      3
Pointer[3] =      0
Pointer[4] =      4
```

Using foreach in global constraints in the following way won't work currently. as the . operator(dot) is not supported in foreach in LRM currently.

## EXAMPLE:

```
class parent;
    rand byte a[0:9];
```

```

endclass
class child_96;
  rand parent P;
  rand byte a[0:9];
  constraint Not_supported { foreach(P.a[i]) P.a[i] == a[i];}
endclass

```

The correct way to write the above constraint is  
**constraint Supported { foreach(a[i]) a[i] == P.a[i];}**

## **CONSTRAINT GUARDS**

Constraint guards are predicate expressions that function as guards against the creation of constraints, and not as logical relations to be satisfied by the solver. These predicate expressions **&&**, **||** and **!** are evaluated before the constraints are solved. This enables users to write constraints that avoid errors due to nonexistent object handles or array indices out of bounds.

There are 4 states when a sub expression is evaluated.

- ⌚ 0 FALSE Subexpression evaluates to FALSE.
- ⌚ 1 TRUE Subexpression evaluates to TRUE.
- ⌚ E ERROR Subexpression causes an evaluation error like null pointer.
- ⌚ R RANDOM Expression includes random variables and cannot be evaluated.

If any of subexpression results ERROR, then randomization fails.

### **EXAMPLE:1**

```

class SList_97;
  rand int n;
  rand Slist_97 next;
  constraint sort { n < next.n; }
endclass

```

In the Example 1, while sorting the elements of array in ascending order, if next is null i.e for last element randomization fails.

### **EXAMPLE:2**

```

class SList_98;
  rand int n;
  rand Slist_98 next;
  constraint sort { if( next != null ) n < next.n; }
endclass

```

In Example 2, Even if next is null, constraint wont be generated so randomization will never fail.

### **EXAMPLE:3**

```

class D;
  int x;
endclass
class C;
  rand int x, y;

```

```

D a, b;
constraint c1 { (x < y || a.x > b.x || a.x == 5 ) -> x+y == 10; }
endclass

```

In Example 3, the predicate subexpressions are  $(x < y)$ ,  $(a.x > b.x)$ , and  $(a.x == 5)$ , which are all connected by disjunction. Some possible cases are as follows:

⌚ Case 1: a is non-null, b is null, a.x is 5.

Because  $(a.x == 5)$  is true, the fact that  $b.x$  generates an error does not result in an error.

The unconditional constraint  $(x+y == 10)$  is generated.

⌚ Case 2: a is null.

This always results in error, irrespective of the other conditions.

⌚ Case 3: a is non-null, b is non-null, a.x is 10, b.x is 20.

All the guard subexpressions evaluate to FALSE.

The conditional constraint  $(x < y) \rightarrow (x+y == 10)$  is generated.

#### EXAMPLE:4

```

class D;
  int x;
endclass
class C;
  rand int x, y;
  D a, b;
  constraint c1 { (x < y && a.x > b.x && a.x == 5 ) -> x+y == 10; }
endclass

```

In Example 4, the predicate subexpressions are  $(x < y)$ ,  $(a.x > b.x)$ , and  $(a.x == 5)$ , which are all connected by conjunction. Some possible cases are as follows:

⌚ Case 1: a is non-null, b is null, a.x is 6.

Because  $(a.x == 5)$  is false, the fact that  $b.x$  generates an error does not result in an error.

The constraint is eliminated.

⌚ Case 2: a is null

This always results in error, irrespective of the other conditions.

⌚ Case 3: a is non-null, b is non-null, a.x is 5, b.x is 2.

All the guard subexpressions evaluate to TRUE, producing constraint  $(x < y) \rightarrow (x+y == 10)$ .

#### EXAMPLE:5

```

class D;
  int x;
endclass
class C;
  rand int x, y;

```

```

D a, b;
constraint c1 { (x < y && (a.x > b.x || a.x == 5)) -> x+y == 10; }
endclass

```

In Example 5, the predicate subexpressions are  $(x < y)$  and  $(a.x > b.x \parallel a.x == 5)$ , which are connected by disjunction. Some possible cases are as follows:

⌚ Case 1: a is non-null, b is null, a.x is 5.

The guard expression evaluates to  $(\text{ERROR} \parallel a.x == 5)$ , which evaluates to  $(\text{ERROR} \parallel \text{TRUE})$ . The guard subexpression evaluates to TRUE.

The conditional constraint  $(x < y) \rightarrow (x+y == 10)$  is generated.

⌚ Case 2: a is non-null, b is null, a.x is 8.

The guard expression evaluates to  $(\text{ERROR} \parallel \text{FALSE})$  and generates an error.

⌚ Case 3: a is null

This always results in error, irrespective of the other conditions.

⌚ Case 4: a is non-null, b is non-null, a.x is 5, b.x is 2.

All the guard subexpressions evaluate to TRUE.

The conditional constraint  $(x < y) \rightarrow (x+y == 10)$  is generated.

#### EXAMPLE:6

```

class A_108;
  rand integer arr[];
  constraint c { foreach( arr[i]) arr[i] == arr[i+1] ;}
endclass

```

In Example 6, generates an error when i is the last element in the array.\

#### TITBITS

The first constraint program which wrote randomized sucessfully but the results are not what expected.

My constraint is to limit Var between 0 and 100.

#### EXAMPLE:

```

class Base;
  rand integer Var;
  constraint randge { 0< Var < 100 ;}
endclass
program inhe_109;
  Base obj;

```

```

initial
begin
    obj = new();
    for(int i=0 ; i < 100 ; i++)
        if(obj.randomize())
            $display(" Randomization successful : Var = %0d ",obj.Var);
        else
            $display("Randomization failed");
    end
endprogram

```

## **RESULTS:**

Randomization successful : Var = 2026924861  
 Randomization successful : Var = -1198182564  
 Randomization successful : Var = 1119963834  
 Randomization successful : Var = -21424360  
 Randomization successful : Var = -358373705  
 Randomization successful : Var = -345517999  
 Randomization successful : Var = -1435493197  
 ..etc

The mistake what I have done is simple, this resulted the constraint solver to solve the statement  $((0 < \text{Var}) < 100)$

For the above equation, are the results are correct.

Then I changed the constraint to {  $0 < \text{Var} ; \text{Var} < 100$  ; }

The solver considered the this constraint as  $(0 < \text{Var}) \&\& (\text{Var} < 100)$ ; and solution are correct.

To generate random values less then -10, the following may not work.

## **EXAMPLE:**

```

class Base_110;
  rand integer Var;
  constraint randge { Var + 10 <= 0 ; }
endclass

```

## **RESULTS:**

Randomization sucessfull : Var = -1601810394  
 Randomization sucessfull : Var = 2147483646  
 Randomization sucessfull : Var = -335544322

$\text{Var} = 2147483646$  is not less -10, but the solver solved it using  $((\text{Var} + 10) \leq 0)$  i.e  $((2147483646 + 10) \leq 0)$  which is true

To solve this use the inside operator.

```
constraint randge { Var inside {[ -10000 : -10 ]}; }
```

Make sure that constraint expression are not mixed up with signed and unsigned variables.

```

# Ran_Stb_1.Var : 4
# Ran_Stb_1.Var : 5
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 1
# Ran_Stb_1.Var : 0
# Ran_Stb_1.Var : 2
# Ran_Stb_1.Var : 2
# Ran_Stb_1.Var : 7
# Ran_Stb_1.Var : 6
# Ran_Stb_1.Var : 0

```

### **Constraining Non Integral Data Types:**

Constraints can be any SystemVerilog expression with variables and constants of integral type (e.g.,

bit, reg, logic, integer, enum, packed struct).

To Constraint a real number, randomize integer and convert it to real as it is required.

#### **EXAMPLE:**

```

class cls;
  rand integer Var;
endclass

class real_c;
  real r;
  rand integer i;
  rand integer j;

  function void post_randomize;
    r = $bitstoreal({i,j});
    $display("%e",r);
  endfunction
endclass

program real_p_111;

  real_c obj = new();
  initial
    repeat(5)
      void'(obj.randomize());
endprogram

```

#### **RESULT:**

2.507685e-280

-1.188526e-07

9.658227e-297

-2.912335e+247

2.689449e+219

### **Saving Memory**

In packet protocol application like PCI Express, the packets which are driven to the DUV has to be manipulated and stored to compare with the actual packet which is coming from DUV. If the packet size is large and number of packets are huge, it occupies more memory. In verilog in this case the whole packet need to be stored. HVL has more advantages w.r.t this case. We can store high level information like packet size, CRC error, header. But functional verification needs to store the payload for checking that the payload did not get corrupted. Major part of the storage taken by the payload itself. If we can avoid storing the payload, we can save lot of storage space. The following technique assigns predictable random values to payload fields. Only the start of the payload need to be stored.

**EXAMPLE:**

```
integer pkt_length;
byte payload[0:MAX];
task gen_payload(integer seed ,integer length);
integer temp_seed;
temp_seed = seed;
for(int i=0;i< length;i++)
begin
temp_seed = $random(temp_seed);
payload[i] = temp_seed;
end
endtask
```

This is the task which checks whether payload is received did not get corrupted.

**EXAMPLE:**

```
task check_payload(integer seed,integer length);
integer temp_seed;
temp_seed = seed;
for(int i=0;i< length;i++)
begin
temp_seed = $random(temp_seed);
if(payload[i] != temp_seed)
$display(" ERROR :: DATA MISMATCH ");
end
endtask
```

## **INTRODUCTION**

## Systemverilog Functional Coverage Features

- ⌚ Coverage of variables and expressions
- ⌚ Cross coverage
- ⌚ Automatic and user-defined coverage bins
  - Values, transitions, or cross products
- ⌚ Filtering conditions at multiple levels
- ⌚ Flexible coverage sampling
  - Events, Sequences, Procedural
- ⌚ Directives to control and query coverage

### **COVER GROUP**

The covergroup construct encapsulates the specification of a coverage model. Each covergroup specification can include the following components:

A clocking event that synchronizes the sampling of coverage points

- ⌚ A set of coverage points
- ⌚ Cross coverage between coverage points
- ⌚ Optional formal arguments
- ⌚ Coverage options

A Cover group is defined between key words **covergroup** & **endgroup**.

A Covergroup Instance can be created using the **new()** operator.

**covergroup** cg;

...  
...  
...

**endgroup**

cg cg\_inst = **new**;

The above example defines a covergroup named "cg". An instance of "cg" is declared as "cg\_inst" and created using the "new" operator.

### **SAMPLE**

Coverage should be triggered to sample the coverage values. Sampling can be done using

- ⌚ Any event expression -edge, variable
- ⌚ End-point of a sequence
- ⌚ Event can be omitted
- ⌚ Calling sample() method.

**covergroup** cg **@(posedge** clk);

...  
...  
...

**endgroup**

The above example defines a covergroup named "cg". This covergroup will be automatically sampled each time there is a posedge on "clk" signal.

```
covergroup cg;
  ...
  ...
  ...
endgroup

cg cg_inst = new;
initial // or task or function or always block
begin
  ...
  ...
  ...
  ...
  ...
end
```

Sampling can also be done by calling explicitly calling .sample() method in procedural code. This is used when coverage sampling is required based on some calculations rather than events.

### **COVER POINTS**

A covergroup can contain one or more coverage points. A coverage point can be an integral variable or an integral expression. A coverage point creates a hierarchical scope, and can be optionally labeled. If the label is specified then it designates the name of the coverage point.

```
program main;
  bit [0:2] y;
  bit [0:2] values[$]={3,5,6};
  covergroup cg;
    cover_point_y : coverpoint y;
  endgroup
  cg cg_inst = new();
  initial
    foreach(values[i])
      begin
        y = values[i];
        cg_inst.sample();
      end
end
```

**endprogram** In the above example, we are sampleing the cover point "y". The cover point is

named "cover\_point\_y" . In the Coverage report you will see this name. A cover group "cg" is defined and its instance "cg\_inst" is created. The value of "y" is sampled when cg\_inst.sample() method is called. Total possible values for Y are 0,1,2,3,4,5,6,7. The variable "y" is assigned only values 3,5,6. The coverage engine should report that only 3 values are covered and there are 8 possible values.

### **Commands To Simulate And Get The Coverage Report:**

#### **⌚ VCS**

Compilation command:

```
vcs -sverilog -ntb_opts dtm filename.sv
```

Simulation Command:

```
./simv
```

Command to generate Coverage report: Coverage report in html format will be in the ./urgReport directory

```
urg -dir simv.vdb
```

#### **⌚ NCSIM**

```
ncverilog -sv -access +rwc -coverage functional filename.sv
```

```
iccr iccr.cmd
```

```
iccr.cmd
```

```
load_test *
```

```
report_detail -instance -both -d *
```

#### **⌚ QUESTASIM**

Create the library

```
vlib library_name
```

Compilation command

```
vlog work library_name filename.sv
```

Simulation Command:

```
vsim library_name.module_top_name
```

Coverage will be saved in UCDB Format in Questasim

Case 1) By default in modelsim.ini file Name of UCDB file will be there, If it is there, it will create one file filename.ucdb

Case 2) Sometimes in modelsim.ini file UCDB File name will be commented in that case we

have to save UCDB File explicitly after vsim command

```
Coverage save filename.ucdb
```

Once u are ready with UCDB File u need to generate coverage report from ucdb file

To generate only Functional coverage report

```
vcover cvg myreport.txt outfilename.ucdb
```

After running the above program, the coverage report will show,

VARIABLE : cover\_point\_y

Expected : 8

Covered : 3

Percent: 37.50.

In the above report, the coverage percentage is calculated by Covered/Expected.

## **COVERPOINT EXPRESSION**

### **Coverpoint Expression**

A coverage point can be an integral variable or an integral Expression.

SystemVerilog allows specifying the cover points in various ways.

④ 1)Using XMR

Example:

```
Cover_xmr : coverpoint top.DUT.Submodule.bus_address;
```

④ 2)Part select

Example:

```
Cover_part: coverpoint bus_address[31:2];
```

④ 3)Expression

Example:

```
Cocver_exp: coverpoint (a*b);
```

④ 4)Function return value

Example:

```
Cover_fun: coverpoint funcation_call();
```

④ 5)Ref variable

Example:

```
covergroup (ref int r_v) cg;
```

```
    cover_ref: coverpoint r_v;
```

```
endgroup
```

## Coverage Filter

The expression within the iff construct specifies an optional condition that disables coverage for that cover point. If the guard expression evaluates to false at a sampling point, the coverage point is ignored.

For example:

```
covergroup cg;
  coverpoint cp_varib iff(!reset); // filter condition
endgroup
```

In the preceding example, cover point varible "cp\_varib" is covered only if the value reset is low.

## GENERIC COVERAGE GROUPS

Generic coverage groups can be written by passing their traits as arguments to the coverage constructor. This allows creating a reusable coverage group which can be used in multiple places.

For example, To cover a array of specified index range.

```
covergroup cg(ref int array, int low, int high ) @(clk);
  coverpoint// sample variable passed by reference
  {
    bins s = { [low : high] };
  }
endgroup

int A, B;
rgc1 = new( A, 0, 50 );// cover A in range 0 to 50
rgc2 = new( B, 120, 600 );// cover B in range 120 to 600
```

The example above defines a coverage group, gc, in which the signal to be sampled as well as the extent of the coverage bins are specified as arguments. Later, two instances of the coverage group are created; each instance samples a different signal and covers a different range of values.

## COVERAGE BINS

A coverage-point bin associates a name and a count with a set of values or a sequence of value transitions. If the bin designates a set of values, the count is incremented every time the coverage point matches one of the values in the set. If the bin designates a sequence of value transitions, the count is incremented every time the coverage point matches the entire sequence of value transitions.

Bins can be created implicitly or explicitly.

## Implicit Bins

While define cover point, if you do not specify any bins, then Implicit bins are created. The number of bins creating can be controlled by auto\_bin\_max parameter.

Example for non enum cover point

```
program main;
    bit [0:2] y;
    bit [0:2] values[$]={3,5,6};
    covergroup cg;
        cover_point_y : coverpoint y
            { option.auto_bin_max = 4; }
    endgroup
    cg cg_inst = new();
    initial
        foreach(values[i])
            begin
                y = values[i];
                cg_inst.sample();
            end
    endprogram
```

In the above example, the auto\_bin\_max is declared as 4. So, the total possible values are divided in 4 parts and each part corresponds to one bin.

The total possible values for variable "y" are 8. They are divided into 4 groups.

Bin[0] for 0 and 1

Bin[1] for 2 and 3

Bin[2] for 4 and 5

Bin[3] for 6 and 7

Variable Y is assigned values 3,5 and 6. Values 3,5 and 6 belongs to bins bin[1],bin[2] and bin[3] respectively. Bin[0] is not covered.

Coverage report:

VARIABLE : cover\_point\_y

Expected : 4

Covered : 3

Percent: 75.00

Uncovered bins

auto[0:1]

Covered bins

```
-----  
auto[2:3]  
auto[4:5]  
auto[6:7]
```

Example of enum data type:

For Enum data type, the numbers of bins are equal to the number of elements of enum data type.  
The bin identifiers are the enum member name.

```
typedef enum { A,B,C,D } alpha;  
program main;  
    alpha y;  
    alpha values[$]={A,B,C};  
    covergroup cg;  
        cover_point_y : coverpoint y;  
    endgroup  
    cg cg_inst = new();  
initial  
    foreach(values[i])  
    begin  
        y = values[i];  
        cg_inst.sample();  
    end  
    endprogram
```

In The above example, the variable "y" is enum data type and it can have 4 enum members A,B,C and D. Variable Y is assigned only 3 Enum members A,B and C.

Coverage report:

```
-----  
VARIABLE : cover_point_y  
Expected : 4  
Covered : 3  
Percent: 75.00
```

Uncovered bins

```
-----  
auto_D
```

Covered bins

```
-----  
auto_C  
auto_B  
auto_A
```

## **EXPLICIT BIN CREATION**

Explicit bin creation is recommended method. Not all values are interesting or relevant in a cover point, so when the user knows the exact values he is going to cover, he can use explicit bins. You can also name the bins.

```
program main;
  bit [0:2] y;
  bit [0:2] values[$] = '{3,5,6};
  covergroup cg;
    cover_point_y : coverpoint y {
      bins a = {0,1};
      bins b = {2,3};
      bins c = {4,5};
      bins d = {6,7};
    }
  endgroup
```

```
cg cg_inst = new();
initial
  foreach(values[i])
    begin
      y = values[i];
      cg_inst.sample();
    end
```

```
endprogram
```

In the above example, bins are created explicitly. The bins are named a,b,c and d.

Coverage report:

```
-----
VARIABLE : cover_point_y
Expected : 4
Covered : 3
Percent: 75.00
```

Uncovered bins

```
-----
a
```

Covered bins

```
-----
b
```

c  
d

## Array Of Bins

To create a separate bin for each value (an array of bins) the square brackets, [], must follow the bin name.

```
program main;
bit [0:2] y;
bit [0:2] values[$]={3,5,6};

covergroup cg;
    cover_point_y : coverpoint y {
        bins a[] = {[0:7]};
    }

endgroup

cg cg_inst = new();
initial
    foreach(values[i])
begin
    y = values[i];
    cg_inst.sample();
end

endprogram
```

In the above example, bin a is array of 8 bins and each bin associates to one number between 0 to 7.

Coverage report:

```
-----
VARIABLE : cover_point_y
Expected : 8
Covered : 3
Percent: 37.50
```

Uncovered bins

```
-----
a_0
a_1
a_2
```

```
a_4  
a_7
```

Covered bins

```
-----  
a_3  
a_5  
a_6
```

To create a fixed number of bins for a set of values, a number can be specified inside the square brackets.

```
program main;  
    bit [0:3] y;  
    bit [0:2] values[$] = '{3,5,6};  
  
    covergroup cg;  
        cover_point_y : coverpoint y {  
            bins a[4] = {[0:7]};  
        }  
  
    endgroup  
  
    cg cg_inst = new();  
    initial  
        foreach(values[i])  
            begin  
                y = values[i];  
                cg_inst.sample();  
            end  
    endprogram
```

In the above example, variable y is 4 bit width vector. Total possible values for this vector are 16.

But in the cover point bins, we have giving the interested range as 0 to 7. So the coverage report is calculated over the range 0 to 7 only. In this example, we have shown the number bins to be fixed to size 4.

Coverage report:

```
-----  
VARIABLE : cover_point_y  
Expected : 4  
Covered : 3  
Percent: 75.00
```

Uncovered bins

---

a[0:1]

Covered bins

---

a[2:3]

a[4:5]

a[6:7]

### **Default Bin**

The default specification defines a bin that is associated with none of the defined value bins. The default bin catches the values of the coverage point that do not lie within any of the defined bins. However, the coverage calculation for a coverage point shall not take into account the coverage captured by the default bin.

```
program main;
  bit [0:3] y;
  bit [0:2] values[$]= '{3,5,6};
  covergroup cg;
    cover_point_y : coverpoint y {
      bins a[2] = {[0:4]};
      bins d = default;
    }
  endgroup
  cg cg_inst = new();
initial
  foreach(values[i])
begin
  y = values[i];
  cg_inst.sample();
end
endprogram
```

In the above example, we have specified only 2 bins to cover values from 0 to 4. Rest of values are covered in default bin <93>d<94> which is not using in calculating the coverage percentage.  
Coverage report:

---

VARIABLE : cover\_point\_y

Expected : 2

Covered : 1

Percent: 50.00

Uncovered bins

-----  
a[0:1]

Covered bins

-----  
a[2:4]

Default bin

-----  
d

### **TRANSITION BINS**

Transitional functional point bin is used to examine the legal transitions of a value.

SystemVerilog allows to specifies one or more sets of ordered value transitions of the coverage point.

Type of Transitions:

- ⌚ Single Value Transition
- ⌚ Sequence Of Transitions
- ⌚ Set Of Transitions
- ⌚ Consecutive Repetitions
- ⌚ Range Of Repetition
- ⌚ Goto Repetition
- ⌚ Non Consecutive Repetition

### **Single Value Transition**

Single value transition is specified as:

value1 => value2

```
program main;  
    bit [0:3] y;  
    bit [0:2] values[$]={3,5,6};
```

```
covergroup cg;  
    cover_point_y : coverpoint y {  
        bins tran_34 =(3=>4);  
        bins tran_56 =(5=>6);  
    }
```

```
endgroup
```

```
cg cg_inst = new();
```

```

initial
  foreach(values[i])
    begin
      y = values[i];
      cg_inst.sample();
    end
  endprogram

```

In the above example, 2 bins are created for covering the transition of point "y" from 3 to 4 and other for 5 to 6. The variable y is given the values and only the transition 5 to 6 is occurring.

Coverage report:

-----  
VARIABLE : cover\_point\_y

Expected : 2

Covered : 1

Percent: 50.00

Uncovered bins

-----  
tran\_34

Covered bins

-----  
tran\_56

### **Sequence Of Transitions**

A sequence of transitions is represented as:

value1 => value3 => value4 => value5

In this case, value1 is followed by value3, followed by value4 and followed by value5. A sequence can be  
of any arbitrary length.

```

program main;
  bit [0:3] y;
  bit [0:2] values[$]= '{3,5,6};

  covergroup cg;
    cover_point_y : coverpoint y {
      bins tran_345 = (3=>4>=5);
      bins tran_356 = (3=>5=>6);
    }

  endgroup

```

```

cg cg_inst = new();
initial
  foreach(values[i])
begin
  y = values[i];
  cg_inst.sample();
end
endprogram

```

In the above example, 2 bins are created for covering the transition of point "y" from 3 to 4 to 5 and other for 3 to 5 to 6. The variable y is given the values and only the transition 3 to 5 to 6 is occurring.

Coverage report:

```

-----
VARIABLE : cover_point_y
Expected : 2
Covered : 1
Percent: 50.00

```

Uncovered bins

```

-----
tran_345

```

Covered bins

```

-----
tran_356

```

### Set Of Transitions

A set of transitions can be specified as:

range\_list1 => range\_list2

```

program main;
  bit [0:3] y;
  bit [0:2] values[$]={3,5,6};

  covergroup cg;
    cover_point_y : coverpoint y {
      bins trans[] = (3,4=>5,6);
    }
  endgroup

```

```

cg cg_inst = new();
initial
  foreach(values[i])
    begin
      y = values[i];
      cg_inst.sample();
    end
  endprogram

```

In the above example, bin trans creates 4 bin for covering 3=>5,4=>5,3=>6 and 4=>6.

Coverage report:

```

-----
VARIABLE : cover_point_y
Expected : 4
Covered : 1
Percent: 25.00

```

Uncovered bins

```

-----
tran_34_to_56:3->6
tran_34_to_56:4->5
tran_34_to_56:4->6

```

Covered bins

```

-----
tran_34_to_56:3->5

```

### Consecutive Repetitions

Consecutive repetitions of transitions are specified using  
trans\_item [\* repeat\_range ]

Here, trans\_item is repeated for repeat\_range times. For example,

3 [\* 5]

is the same as

3=>3=>3=>3=>3

**program** main;

**bit** [0:3] y;

**bit** [0:2] values[\$]= '{3,3,3,4,4};

**covergroup** cg;

**cover\_point\_y** : **coverpoint** y {

**bins** trans\_3 = (3[\*5]);

```
bins trans_4 = (4[*2]);  
}
```

**endgroup**

```
cg cg_inst = new();  
initial  
foreach(values[i])  
begin  
    y = values[i];  
    cg_inst.sample();  
end
```

**endprogram**

Coverage report:

```
-----  
VARIABLE : cover_point_y  
Expected : 2  
Covered : 1  
Percent: 50.00
```

Uncovered bins

```
-----  
trans_3
```

Covered bins

```
-----  
trans_4
```

### Range Of Repetition

An example of a range of repetition is:

3 [\* 3:5]

is the same as

3=>3=>3, 3=>3=>3=>3, 3=>3=>3=>3=>3

```
program main;  
bit [0:3] y;  
bit [0:2] values[$]= '{4,5,3,3,3,3,6,7};
```

```
covergroup cg;  
    cover_point_y : coverpoint y {
```

```
bins trans_3[] = (3[*3:5]);  
}
```

### **endgroup**

```
cg cg_inst = new();  
initial  
  foreach(values[i])  
    begin  
      y = values[i];  
      cg_inst.sample();  
    end
```

### **endprogram**

In the above example, only the sequence 3=>3=>3=>3 is generated. Other expected sequences 3=>3=>3 and 3=>3=>3=>3 are not generated.

Coverage report:

```
-----  
VARIABLE : cover_point_y  
Expected : 3  
Covered : 1  
Percent: 33.33
```

Uncovered bins

```
-----  
tran_3:3[*3]  
tran_3:3[*5]
```

Covered bins

```
-----  
tran_3:3[*4]
```

### **Goto Repetition**

The goto repetition is specified using: trans\_item [-> repeat\_range]. The required number of occurrences of a particular value is specified by the repeat\_range. Any number of sample points can occur before the first occurrence of the specified value and any number of sample points can occur between each occurrence of the specified value. The transition following the goto repetition must immediately follow the last occurrence of the repetition.

For example:

3 [-> 3]

is the same as

....=>3...=>3...=>3

where the dots (...) represent any transition that does not contain the value 3.

A goto repetition followed by an additional value is represented as follows:

1 => 3 [ -> 3] => 5

is the same as

1...=>3...=>3...=>3 =>5

**program** main;

**bit** [0:3] y;

**bit** [0:2] values[\$]={1,6,3,6,3,6,3,5};

**covergroup** cg;

```
cover_point_y : coverpoint y {  
    bins trans_3 = (1=>3[->3]=>5);  
}
```

**endgroup**

cg cg\_inst = new();

**initial**

**foreach**(values[i])

**begin**

    y = values[i];

    cg\_inst.sample();

**end**

**endprogram**

Coverage report:

VARIABLE : cover\_point\_y

Expected : 1

Covered : 1

Percent: 100.00

### Non Consecutive Repetition

The nonconsecutive repetition is specified using: trans\_item [= repeat\_range]. The required number of occurrences of a particular value is specified by the repeat\_range. Any number of sample points can occur before the first occurrence of the specified value and any number of sample points can occur between each occurrence of the specified value. The transition following the nonconsecutive repetition may occur after any number of sample points so long as the repetition value does not occur again.

For example:

3 [= 2]

is same as

...=>3...=>3

A nonconsecutive repetition followed by an additional value is represented as follows:

1 => 3 [=2] => 5

is the same as

1...=>3...=>3...=>5

```
program main;
  bit [0:3] y;
  bit [0:2] values[$]={1,6,3,6,3,6,5};
  covergroup cg;
    cover_point_y : coverpoint y {
      bins trans_3 =(1=>3[=2]=>5);
    }
  endgroup
  cg cg_inst = new();
  initial
    foreach(values[i])
      begin
        y = values[i];
        cg_inst.sample();
      end
  endprogram
```

Coverage report:

VARIABLE : cover\_point\_y

Expected : 1

Covered : 1

Percent: 100.00

### **WILDCARD BINS**

By default, a value or transition bin definition can specify 4-state values.

When a bin definition includes an X or Z, it indicates that the bin count should only be incremented when the sampled value has an X or Z in the same bit positions.

The wildcard bins definition causes all X, Z, or ? to be treated as wildcards for 0 or 1 (similar to the ==? operator).

For example:

```
wildcard bins g12_16 = { 4'b11?? };
```

The count of bin g12\_16 is incremented when the sampled variable is between 12 and 16:

```
1100 1101 1110 1111
program main;
reg [0:3] y;
reg [0:3] values[$]={ 4'b1100,4'b1101,4'b1110,4'b1111};
covergroup cg;
cover_point_y : coverpoint y {
    wildcard bins g12_15 = { 4'b11?? } ;
}
endgroup
cg cg_inst = new();
initial
foreach(values[i])
begin
    y = values[i];
    cg_inst.sample();
end
endprogram
```

Coverage report:

-----  
VARIABLE : cover\_point\_y

Expected : 1

Covered : 1

Percent: 100.00

Covered bin

-----  
g12\_15

Number of times g12\_15 hit : 4

Similarly, transition bins can define wildcard bins.

For example:

wildcard bins T0\_3 =(2'b0x => 2'b1x);

The count of transition bin T0\_3 is incremented for the following transitions (as if by (0,1=>2,3)):

00 => 10 , 00 => 11, 01 => 10 , 01 => program main;

```

reg [0:1] y;
reg [0:1] values[$]={ 2'b00,2'b01,2'b10,2'b11};

covergroup cg;
    cover_point_y : coverpoint y {
        wildcard bins trans = (2'b0X => 2'b1X );
    }
    endgroup

cg cg_inst = new();
initial
    foreach(values[i])
    begin
        y = values[i];
        cg_inst.sample();
    end

endprogram

```

Coverage report:

```

-----
VARIABLE : cover_point_y
Expected : 1
Covered : 1
Percent: 100.00

```

Covered bin

```

-----
trans
Number of times trans hit : 1 (01 => 10)
IGNORE BINS

```

A set of values or transitions associated with a coverage-point can be explicitly excluded from coverage by specifying them as ignore\_bins.

```

program main;
    bit [0:2] y;
    bit [0:2] values[$]={1,6,3,7,3,4,3,5};

    covergroup cg;
        cover_point_y : coverpoint y {

```

```

ignore_bins ig = {1,2,3,4,5};
}

endgroup
cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end
endprogram

```

In the above program, total possible values for y are 0 to 7. Ignore\_bins specified to Ignored values between 1 to 5. So the Expected values are 0,6 and 7. Out of these expected values, only 6 and 7 are generated.

Coverage report:

```
-----
VARIABLE : cover_point_y
Expected : 3
Covered : 2
Percent: 66.66
```

Uncovered bins

```
-----
auto[0]
Excluded/Illegal bins
```

```
-----
ig
auto[1]
auto[2]
auto[3]
auto[4]
auto[5]
```

Covered bins

```
-----
auto[6]
auto[7]
```

## **ILLEGAL BINS**

A set of values or transitions associated with a coverage-point can be marked as illegal by specifying them as illegal\_bins. All values or transitions associated with illegal bins are excluded from coverage. If an illegal value or transition occurs, a runtime error is issued.

```
program main;
  bit [0:2] y;
  bit [0:2] values[$] = '{1,6,3,7,3,4,3,5};

  covergroup cg;
    cover_point_y : coverpoint y {
      illegal_bins ib = {7};
    }
  endgroup

  cg cg_inst = new();
  initial
    foreach(values[i])
      begin
        y = values[i];
        cg_inst.sample();
      end
    endprogram
```

Result:

```
-----
** ERROR **

Illegal state bin ib of coverpoint cover_point_y in
covergroup cg got hit with value 0x7
```

## **CROSS COVERAGE**

Cross allows keeping track of information which is received simultaneous on more than one cover point. Cross coverage is specified using the cross construct.

```
program main;
  bit [0:1] y;
  bit [0:1] y_values[$] = '{1,3};

  bit [0:1] z;
  bit [0:1] z_values[$] = '{1,2};

  covergroup cg;
    cover_point_y : coverpoint y ;
    cover_point_z : coverpoint z ;
```

```

cross_yz : cross cover_point_y,cover_point_z ;
endgroup

cg cg_inst = new();
initial
  foreach(y_values[i])
    begin
      y = y_values[i];
      z = z_values[i];
      cg_inst.sample();
    end

endprogram

```

In the above program, y has can have 4 values 0,1,2 and 3 and similarly z can have 4 values 0,1,2 and 3. The cross product of the y and z will be 16 values (00),(01),(02),(03),(10),(11).....(y,z).....(3,2)(3,3) . Only combinations (11) and (32) are generated.  
Cross coverage report: cover points are not shown.

Covered bins

---

```

cover_point_y  cover_point_z
auto[3]        auto[2]
auto[1]        auto[1]

```

### User-Defined Cross Bins

User-defined bins for cross coverage are defined using bin select expressions.

Consider the following example code:

```

int i,j;
covergroup ct;
  coverpoint i { bins i[] = { [0:1] }; }
  coverpoint j { bins j[] = { [0:1] }; }
  x1: cross i,j;
  x2: cross i,j {
    bins i_zero = binsof(i) intersect { 0 };
  }
endgroup

```

Cross x1 has the following bins:

```
<i[0],j[0]>  
<i[1],j[0]>  
<i[0],j[1]>  
<i[1],j[1]>
```

Cross x2 has the following bins:

```
i_zero  
<i[1],j[0]>  
<i[1],j[1]>
```

## **COVERAGE OPTIONS**

Options control the behavior of the covergroup, coverpoint, and cross.

There are two types of options:

- ⌚ those that are specific to an instance of a covergroup and
- ⌚ those that specify an option for the covergroup type as a whole.

### **Weight**

Syntax : weight= number

default value: 1

Description :

If set at the covergroup syntactic level, it specifies the weight of this covergroup instance for computing the overall instance coverage of the simulation. If set at the coverpoint (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the instance coverage of the enclosing covergroup. The specified weight shall be a non-negative integral value.

### **Goal**

Syntax :goal=number

default value: 100

Description :

Specifies the target goal for a covergroup instance or for a coverpoint or a cross of an instance.

### **Name**

Syntax :name=string

default value:unique name

Description :

Specifies a name for the covergroup instance. If unspecified, a unique name for each instance is automatically generated by the tool.

## **Comment**

Syntax :comment=string

default value: ""

Description :

A comment that appears with a covergroup instance or with a coverpoint or cross of the covergroup instance. The comment is saved in the coverage database and included in the coverage report.

## **At least**

Syntax :at\_least=number

default value: 1

Description :

Minimum number of hits for each bin. A bin with a hit count that is less than number is not considered covered.

## **Detect overlap**

Syntax :detect\_overlap=Boolean

default value: 0

Description :

When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a coverpoint.

## **Auto bin max**

Syntax :auto\_bin\_max=number

default value: 64

Description :

Maximum number of automatically created bins when no bins are explicitly defined for a coverpoint.

## **Cross num print missing**

Syntax :cross\_num\_print\_missing=number

default value: 0

Description :

Number of missing (not covered) cross product bins that shall be saved to the coverage database and printed in the coverage report.

## **Per instance**

Syntax :per\_instance=Boolean

default value: 0

Description :

Each instance contributes to the overall coverage information for the covergroup type. When true, coverage information for this covergroup instance shall be saved in the coverage database and included in the coverage report. When false, implementations are not required to save instance-specific information.

## Get inst coverage

Syntax :get\_inst\_coverage=Boolean

default value: 0

Description :

Only applies when the merge\_instances type option is set . Enables the tracking of per instance coverage with the get\_inst\_coverage built-in method. When false, the value returned by get\_inst\_coverage shall equal the value returned by get\_coverage

Following Table summarizes the syntactical level (covergroup, coverpoint, or cross) in which type options can be specified.

Option	covergroup	coverpoint	cross
Weight	yes	yes	yes
Goal	yes	yes	yes
Comment	yes	yes	yes
Strobe	yes	no	no

© www.testbench.in

## COVERAGE METHODS

The following coverage methods are provided for the covergroup. These methods can be invoked procedurally at any time.

© void sample():

Description : Triggers sampling of the covergroup

© real get\_coverage()

real get\_coverage(ref int, ref int)

Description : Calculates type coverage number (0...100)

The get\_coverage() method returns the cumulative (or type) coverage, which considers the contribution of all instances of a particular coverage item. and it is a static method that is available on both types (via the :: operator) and instances (using the "." operator).

The get\_coverage() method both accept an optional set of arguments, a pair of int values passed by reference. When the optional arguments are specified, the get\_coverage() method assign to the first argument the value of the covered bins, and to the second argument the number of bins for the given coverage item. These two values correspond to the numerator and the denominator used for calculating the particular coverage number (i.e., the return value before scaling by 100).

© real get\_inst\_coverage()

real get\_inst\_coverage(ref int, ref int)

Description : Calculates the coverage number (0...100)  
 get\_inst\_coverage() method returns the coverage of the specific instance on which it is invoked, thus, it can only be invoked via the "." operator.

The get\_inst\_coverage() method both accept an optional set of arguments, a pair of int values passed by reference. When the optional arguments are specified, the get\_inst\_coverage() method assign to the first argument the value of the covered bins, and to the second argument the number of bins for the given coverage item. These two values correspond to the numerator and the denominator used for calculating the particular coverage number (i.e., the return value before scaling by 100).

**© void set\_inst\_name(string)**

Description : Sets the instance name to the given string

**© void start()**

Description : Starts collecting coverage information

**© void stop()**

Description : Stops collecting coverage information

Method (function)	Covergroup	Coverpoint	Cross
void sample()	Yes	No	No
real get_coverage()	Yes	Yes	Yes
real get_inst_coverage()	Yes	Yes	Yes
void set_inst_name()	Yes	No	No
void start()	Yes	Yes	Yes
void stop()	Yes	Yes	Yes

© www.testbench.in

## **SYSTEM TASKS**

SystemVerilog provides the following system tasks and functions to help manage coverage data collection.

**© \$set\_coverage\_db\_name ( name ) :**

Sets the filename of the coverage database into which coverage information is saved at the end of a simulation run.

**© \$load\_coverage\_db ( name ) :**

Load from the given filename the cumulative coverage information for all coverage group types.

⌚ \$get\_coverage ( ) :

Returns as a real number in the range 0 to 100 the overall coverage of all coverage group types. This number is computed as described above.

## **COVER PROPERTY**

Cover statement can be used to monitor sequences and other behavioral aspects of the design. The tools can gather information about the evaluation and report the results at the end of simulation. When the property for the cover statement is successful, the pass statements can specify a coverage function, such as monitoring all paths for a sequence. The pass statement shall not include any concurrent assert, assume or cover statement. A cover property creates a single cover point.

Coverage results are divided into two: coverage for properties, coverage for sequences.

For sequence coverage, the statement appears as:

Cover property ( sequence\_expr ) statement\_or\_null

### **Cover Property Results**

The results of coverage statement for a property shall contain:

- ⌚ Number of times attempted
- ⌚ Number of times succeeded
- ⌚ Number of times failed
- ⌚ Number of times succeeded because of vacuity

In addition, statement\_or\_null is executed every time a property succeeds.

Vacuity rules are applied only when implication operator is used. A property succeeds non-vacuously only if the consequent of the implication contributes to the success.

### **Cover Sequence Results**

Results of coverage for a sequence shall include:

- ⌚ Number of times attempted
- ⌚ Number of times matched (each attempt can generate multiple matches)

In addition, statement\_or\_null gets executed for every match. If there are multiple matches at the same time, the statement gets executed multiple times, one for each match.

It is recommended to cover sequences and not properties and its easy to convert a property into a sequence if required.

### **Coverage property can be declared in**

- ⌚ A design or a separate module
- ⌚ Packages

- ⌚ Interfaces
- ⌚ Program block

Cover properties are not allowed in class.

### **Comparison Of Cover Property And Cover Group.**

Cover groups can reference data sets where as cover property references a temporal expression.



Cover group can be triggered using .sample method ()  
Cover property dont have this option.



Cover group has multiple bins options.  
Cover property has only one bin.



Cover group cannot handle complex temporal relationships.  
Cover properties can cover complex temporal expressions.



Cover group automatically handles the crosses.  
Cover properties cannot do crosses.



Cover group has lot of filtering options.  
Cover property has no specific filtering constructs but it can be filtered.



Cover properties cannot be used in classes.  
Cover groups can be used in classes. So, cover groups can reference the variables in class.



Cover groups are most useful at a higher level of abstractions where as cover property makes sense to use when we want to work at low level signals.

We can mix cover group and cover property to gain the OO and temporal advantages. Using properties for temporal expressions and trigger the cover group.

## **INTRODUCTION**

SystemVerilog adds features to specify assertions of a system. An assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and generate input stimulus for validation. The evaluation of the assertions is guaranteed to be equivalent between simulation, which is event-based, and formal verification, which is cycle-based.

SystemVerilog allows assertions to communicate information to the test bench and allows the test bench to react to the status of assertions without requiring a separate application programming interface (API) of any kind

### **Advantages Of Assertion:**

Improving Observability.

Reduces the debug time.

Bugs can be found earlier and are more isolated.

Controllable severity level.

Can interact with C functions.

Describe the Documentation and Specification of the design.

### **What Assertions Can Verify:**

Assertions can be used to capture the information about various level of properties.

conceptual : can be used to verify systemlevel properties which are more architectural level.

design : These expresses unit level properties.

programming: More specified at RTL level.

1)conditional: It checks the some conditional to be true using boolean expressions.

2)sequence : Checks whether the properties arr true using temporal expression.

3)signal : Checks on signal types.

a)x detection :Can be used to detect unconnected ports or undriven signal.

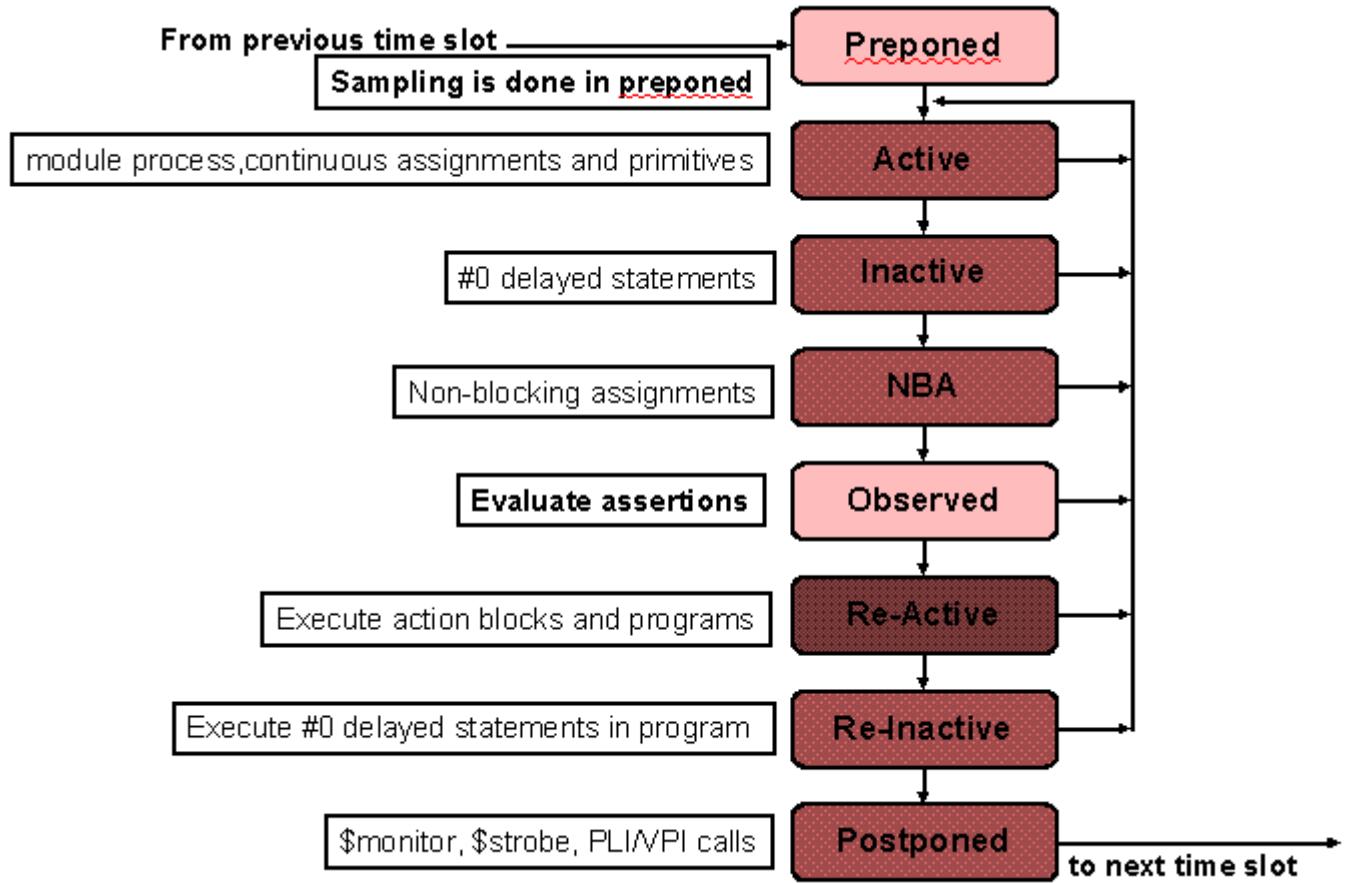
b)encoding types: Checks whether the encoding is violated.

1)onehot

2)gray code

## **EVENT SIMULATION**

The Problem of race condition exists if the SVA is sampling the signals exactly at specified event. So in SystemVerilog , the language has very clear semantics to avoid race condition while evaluating SVA.



In SV, a new region is added before Active region called preponed. So sampling for SVA is done in preponed region. No assignments are not done in preponed region. Signals are stable from previous timeslot and they are occurring before active and NBA ,so the race condition is avoided by this new preponed region. Look at the diagram, regions which are in light cyan color are for SVA.

IN preponed region only sampling is done , the evaluation of these sampled values are done in another region called observed region. Observed region occurs after NBA region. Even though the assignments are done in active ,inactive,NBA region, these updates are not used in observed region. Only signals sample in preponed region are used in observed region. Observed region occurs before reactive region where the testbench executes.

But in immediate assertions, the updated values in previous regions of current time slot are used in observed region.

## ASSERTION TYPES

There are two types of assertions.

Immediate assertions are useful for checking combinational expression. These are similar to if and else statement but with assertion control. Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block.

### Differences between immediate and concurrent assertions

#### • Immediate assertions

- Based on simulation events
- Used only with dynamic simulation tools.
- Used without property keyword
- Placed in procedural block definition

#### • Concurrent assertions

- Based on clock cycles
- Used with both formal and dynamic simulation tools
- Used with property keyword
- Placed in procedural blocks, modules, interfaces or program definitions.

## EXAMPLE:

```
time t;  
always @(posedge clk)  
if (state == REQ)  
assert (req1 || req2);  
else begin  
t = $time;  
#5 $error("assert failed at time %0t",t);  
end
```

## ASSERTION SYSTEM TASKS

Because the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is error. Other severity levels can be specified by including one of the following severity system tasks in the fail statement:

\$fatal is a run-time fatal.

\$error is a run-time error.

\$warning is a run-time warning, which can be suppressed in a tool-specific manner.

\$info indicates that the assertion failure carries no specific severity.

### **Assertion Control System Tasks:**

SystemVerilog provides three system tasks to control assertions.

\$assertoff shall stop the checking of all specified assertions until a subsequent \$asserton. An assertion that is already executing, including execution of the pass or fail statement, is not affected.

\$assertkill shall abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent \$asserton.

\$asserton shall reenable the execution of all specified assertions. When invoked with no arguments, the system task shall apply to all assertions. When the task is specified with arguments, the first argument indicates levels of the hierarchy, consistent with the corresponding argument to the Verilog \$dumpvars system task. Subsequent arguments specify which scopes of the model to control. These arguments can specify entire modules or individual assertions.

### **Boolean System Function:**

\$countones : Returns the numbers of 1's in a bit vector.

\$past : Returns the values of the past.

\$stable : If the Signal is stable, then it returns 1.

\$isunknown : If th X is seen in expression , then it returns 1.

\$rose : returns true if the LSB of the expression changed to 1. Otherwise, it returns false.

\$fell : returns true if the LSB of the expression changed to 0. Otherwise, it returns false.

\$onehot : returns true if only 1 bit of the expression is high.

\$onehot0 : returns true if at most 1 bit of the expression is high.

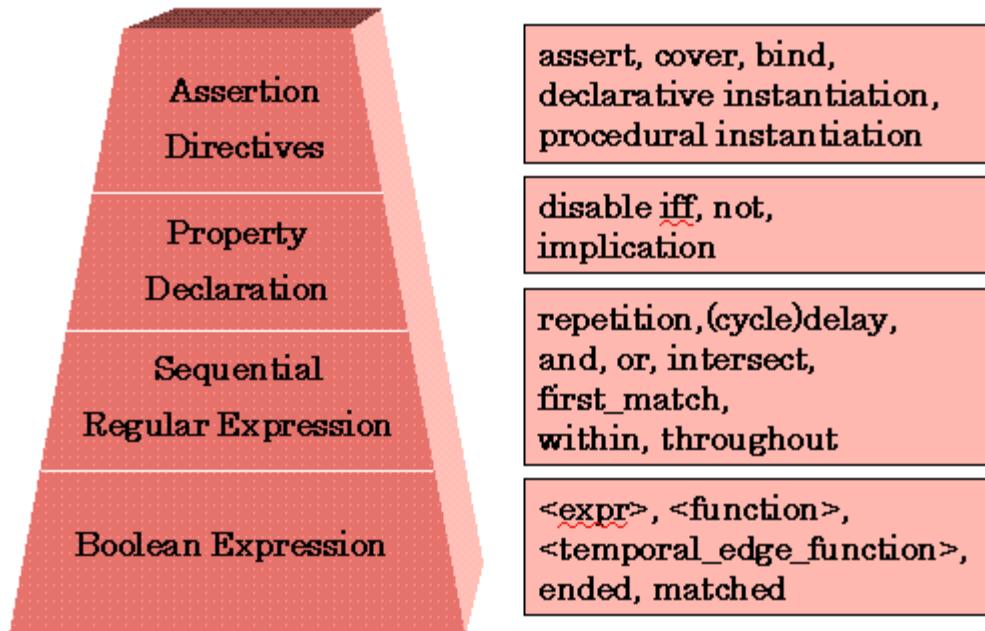
What is the difference between \$rose and posedge?

posedge returns event where as \$rose returns a boolean value. Events cannot be used in expression, \$rose can be used.

### **CONCURRENT ASSERTION LAYERS**

Concurrent assertions describe behavior that spans over time. Unlike immediate assertions, the evaluation model is based on a clock so that a concurrent assertion is evaluated only at the occurrence of a clock tick.

There are 4 layers in concurrent assertions. They are  
 Boolean expressions.  
 Sequence expression.  
 Property declaration.  
 Verification directives.



These layering concept allows to build hierarchical constructs so its easy to maintain them.

### **Boolean Expressions:**

Boolean expression doesn't consume time. The result of the expression is 1,0,x & z. If the result is 1, then the expression is true , else if the expression is 0,x or z , it is false. Concurrent assertions use boolean expressions along with temporal expressions. Immediate assertions use only boolean expressions.

Integral data types such as int,integer,reg,bit,byte,logic,array( elements only),structs,function return values are allowed in boolean expressions. Complex data types like classes,smart quesus, dynamic arrays, associative arrays are not allowed.

### **SEQUENCES**

Boolean logic doenot have the concept of time. Temporal logic is boolean logic with time as one more dimention. Sequence allows us to define temporal nature of the signals using temporal expressions.

Sequences can be composed by concatenation, analogous to a concatenation of lists. The concatenation specifies a delay, using ##, from the end of the first sequence until the beginning of the second sequence.

## indicates cycle delay.

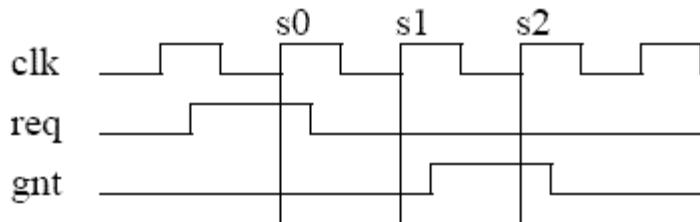
**Fixed Delay:**

A ## followed by a number specifies the delay from the current clock tick to the beginning of the sequence that follows.

**EXAMPLE:**

req ##2 gnt

This specifies that req shall be true on the current clock tick, and gnt shall be true on the second subsequent clock tick



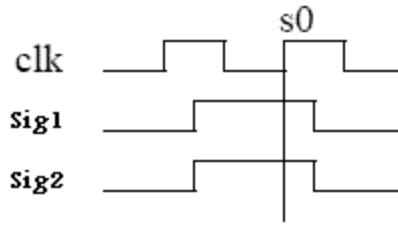
**Zero Delay:**

The delay ##0 indicates that the beginning of the second sequence is at the same clock tick as the end of the first sequence. This can also be achieved using boolean expressions && .

**EXAMPLE:**

sig1 ##0 sig2

This specifies that sig1 shall be true on the current clock tick, and sig2 shall be true on the same clock tick.



### Constant Range Delay:

A `##[n:m]` followed by a range specifies the delay from the current clock tick to the beginning of the sequence that follows.

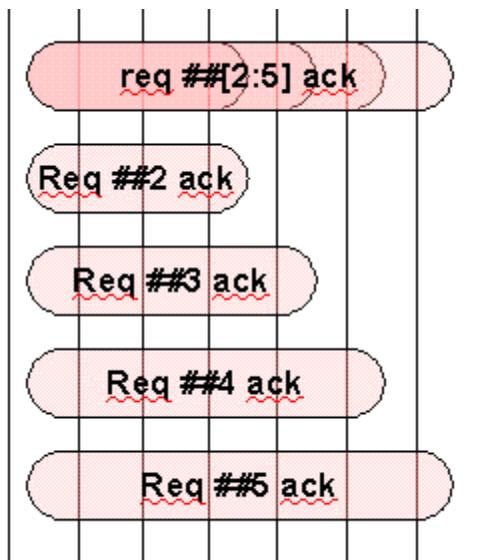
#### **EXAMPLE:**

`req ##[2:5] ack`

This specifies that ack shall be true ,2-5 cycles after req. It creates multiple subsequence threads. This result in multiple hits or fails.

Sub sequences created by range delay in above expressions:

```
req ##2 ack
req ##3 ack
req ##4 ack
req ##5 ack
```



### Unbounded Delay Range:

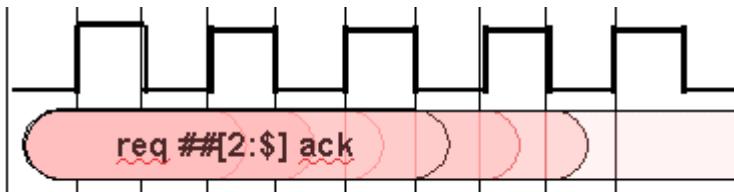
The \$ token is used to indicate the end of simulation.

`##[n:$]` specifies that delay from between n cycles later and till the end of simulation.

### **EXAMPLE:**

```
req##[4:$] ack;
```

This specifies that ack must be true atleast 4 cycles later .



### **Repetition Operators:**

The number of iterations of a repetition can either be specified by exact count or be required to fall within a finite range. If specified by exact count, then the number of iterations is defined by a non-negative integer constant expression. If required to fall within a finite range, then the minimum number of iterations is defined by a non-negative integer constant expression; and the maximum number of iterations either is defined by a non-negative integer constant expression or is \$, indicating a finite, but unbounded, maximum.

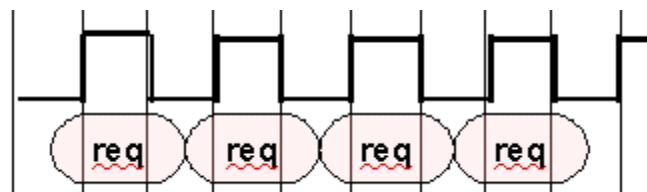
### **Consecutive Repetition:**

Consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand.

### **EXAMPLE:**

```
REQ[*4]
```

This example specifies that ack shell come after req comes 4 times consecutively.



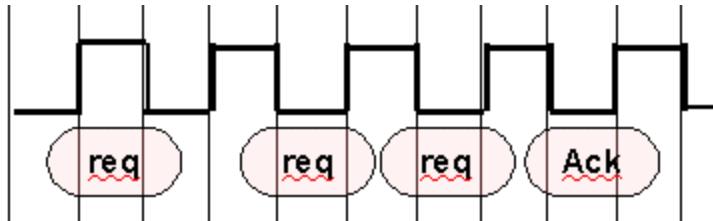
### Goto Repetition :

Goto repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

#### **EXAMPLE:**

req[ $->3$ ]##1 ack

This example specifies that ack shell come after req comes 3 times with no gap between thw last req and ack.

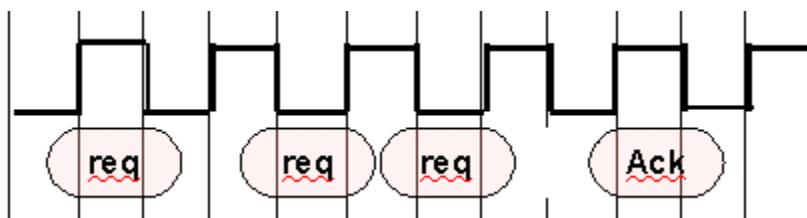


**Nonconsecutive Repetition:** Nonconsecutive repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.

#### **EXAMPLE:**

req[=3]##1 ack

This example specifies that ack shell come after req comes 4 times with gap between thw last req and ack.

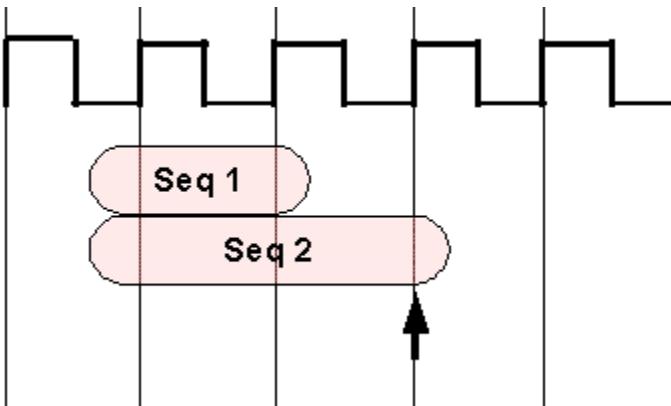


### Sequence And :

Sequence must start and can end at the any time. Match is done after last sequence is ended.

#### EXAMPLE:

Seq1 **and** seq2

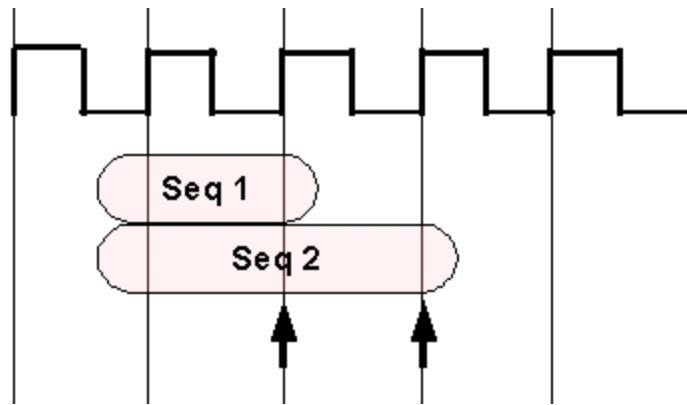


### Sequence Or:

Sequence must start at the same time and can end at any time. Match is done at both the sequences ends.

#### EXAMPLE:

seq1 **or** seq2

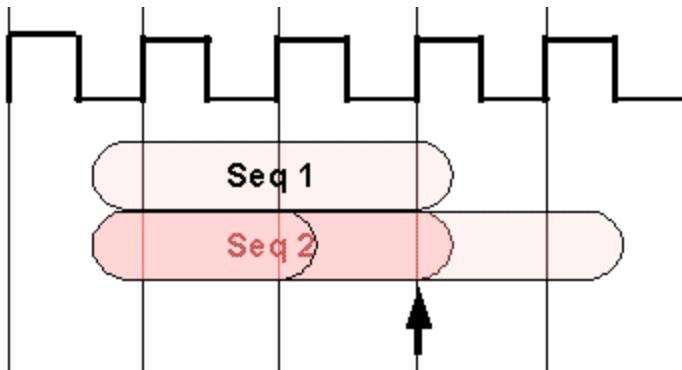


### Sequence Intersect:

sequences must start at the same time and end at same time. Match is done at the end time.

#### EXAMPLE:

Seq1 **intersect** seq2

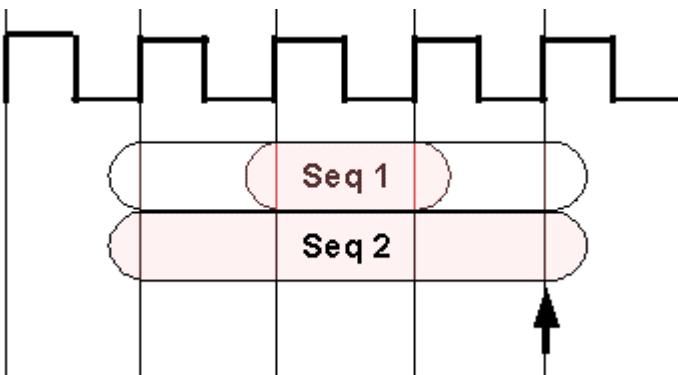


### Sequence Within

One sequence can fully contain another sequence

#### EXAMPLE:

Seq1 **within** seq2

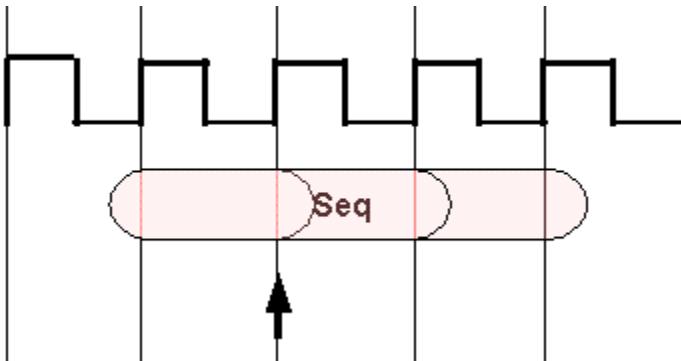


### Sequence First match:

Matches only the first match and ignores other matches.

**EXAMPLE:**

**first\_match(seq1)**

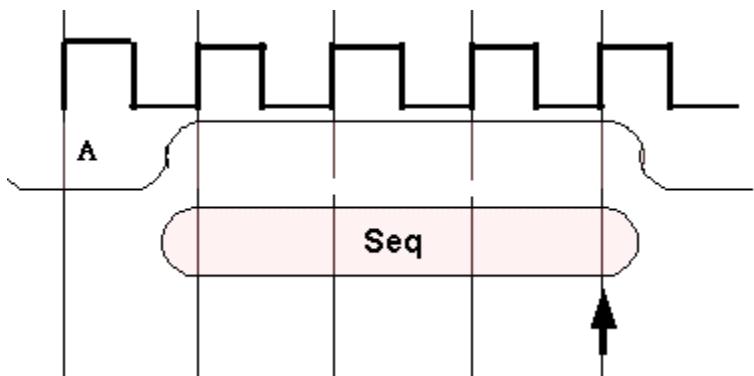


### Sequence Throughout

The throughout operator requires a boolean to be true throughout a sequence.

**EXAMPLE:**

**A throughout seq1**



### Sequence Ended:

The end point of a sequence is reached whenever the ending clock tick of a match of the sequence is reached, regardless of the starting lock tick of the match. The reaching of the end

point can be tested by using the method ended.

**EXAMPLE:**

```
sequence e1;  
@(posedge sysclk) $rose(ready) ##1 proc1 ##1 proc2 ;  
endsequence  
sequence rule;  
@(posedge sysclk) reset ##1 inst ##1 e1.ended ##1 branch_back;  
endsequence
```

**Operator Precedence Associativity:**

Operator precedence and associativity are listed in the following Table . The highest precedence is listed first.

SystemVerilog expression operators	Associativity
[* ] [= ] [-> ]	—
##	Left
throughout	Right
within	Left
intersect	Left
and	Left
or	Left

**PROPERTIES**

A property defines a behavior of the design. A property can be used for verification as an assumption, a checker, or a coverage specification.

Sequences are often used to construct properties. usage of sequences in properties brakes down the complexity. Sequence can be reused across various properties.

A property can be declared in any of the following:

- A module
- An interface
- A program
- A clocking block
- A package
- A compilation-unit scope

Properties constructs:

Disable iff

Implication (if ..else)

overlapping implication ( $|-\rightarrow$ )

Non overlapping implication( $||-\rightarrow$ )

not

**EXAMPLE:**

```
property rule6_with_type(bit x, bit y);
    ##1 x |-> ##[2:10] y;
    //antecedent |-> consequent
endproperty
```

The left-hand operand sequence\_expr is called the antecedent, while the right-hand operand property\_expr is called the consequent.

if antecedent is false, then consequent is not cared and property is considered as vacuous success.

if antecedent is True and if consequent is false then property is considered as false.

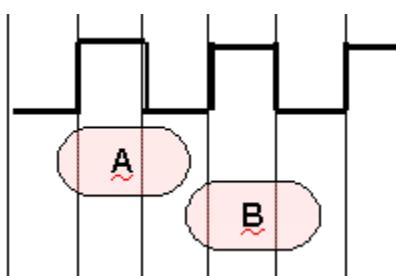
if antecedent is True and if consequent is true then property is considered as true.

**Overlap Implication:**

Consequent expression is evaluated on the same clock of antecedent.

**EXAMPLE:**

a |-> b

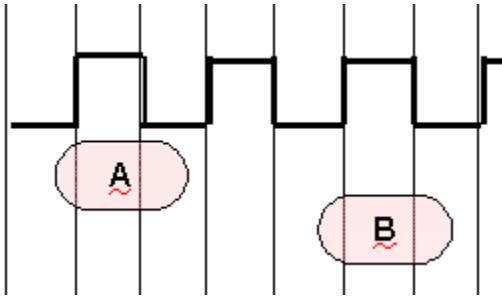


**Non Overlapping Implication**

Consequent expression is evaluated on the next clock of antecedent

## **EXAMPLE:**

a  $\parallel\rightarrow$  b



## **VERIFICATION DIRECTIVE**

A property on its own is never evaluated for checking an expression. It must be used within a verification statement for this to occur. A verification statement states the verification function to be performed on the property.

The statement can be one of the following:

- 1) assert to specify the property as a checker to ensure that the property holds for the design
- 2) assume to specify the property as an assumption for the environment
- 3) cover to monitor the property evaluation for coverage

A concurrent assertion statement can be specified in any of the following:

- 1) An always block or initial block as a statement, wherever these blocks can appear
- 2) A module
- 3) An interface
- 4) A program

**Assert:** The assert statement is used to enforce a property as a checker. When the property for the assert statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the action\_block are executed.

## **EXAMPLE:**

```
a1_assertion:assert property ( @(posedge clk) req inside {0, 1} );
property proto_assertion ;
@(posedge clk) req |> req[*1:$] ##0 ack;
endproperty
```

## **Assume:**

The environment must be constrained so that the properties that are assumed shall hold. Like an assert property, an assumed property must be checked and reported if it fails to hold. There is no requirement on the tools to report successes of the assumed properties.

### **EXAMPLE:**

```
a1:assume property ( @(posedge clk) req dist {0:=40, 1:=60} );
property proto ;
@(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

### **Cover Statement:**

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax is used with the cover statement. The tools can gather information about the evaluation and report the results at the end of simulation. When the property for the cover statement is successful, the pass statements can specify a coverage function, such as monitoring all paths for a sequence. The pass statement shall not include any concurrent assert, assume, or cover statement.

Coverage results are divided into two categories: coverage for properties and coverage for sequences.

Coverage for sequences:

- Number of attempts
- Number of matches
- Multiple matches per attempt are all counted

Coverage for properties:

- Number of attempts
- Number of passes
- Number of vacuous passes
- Number of failures

### **Expect Statement:**

The expect statement is a procedural blocking statement that allows waiting on a property evaluation. The syntax of the expect statement accepts a named property or a property declaration. The expect statement accepts the same syntax used to assert a property. An expect statement causes the executing process to block until the given property succeeds or fails. The statement following the expect is scheduled to execute after processing the Observe region in which the property completes its evaluation. When the property succeeds or fails, the process unblocks, and the property stops being evaluated

### **EXAMPLE:**

```
program tst;
initial begin
# 200ms;
expect( @(posedge clk) a ##1 b ##1 c ) else $error( "expect failed" );
ABC: ...
```

```
end  
endprogram
```

### **Binding:**

To facilitate verification separate from design, it is possible to specify properties and bind them to specific modules or instances. The following are some goals of providing this feature:

It allows verification engineers to verify with minimum changes to the design code and files.

It allows a convenient mechanism to attach verification intellectual Protocol (VIP) to a module or an instance.

No semantic changes to the assertions are introduced due to this feature. It is equivalent to writing properties external to a module, using hierarchical path names.

The bind directive can be specified in any of the following:

A module

An interface

A compilation-unit scope

## **INTRODUCTIONS**

### **What Is Dpi-C ?**

From long time , Users have needs a simple way of communication to foreign languages from verilog. VPI and PLI are not easy interfaces to Use . Users need detailed knowledge of PLI and VPI even for a simple program. Most of the time, users do not need the sophisticated capabilities of VPI and PLI. DPI also permits C/C++ code to wait for Verilog events and C/C++ tasks and functions can be disabledfrom SystemVerilog.

SystemVerilog introduces a new foreign language interface called the Direct Programming Interface (DPI). The DPI provides a very simple, straightforward, and efficient way to connect SystemVerilog and foreign language code unlike PLI or VPI.

DPI developed based on the donations from Synopsys "DirectC interface".

DPI consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI-C are fully isolated. Which programming language is actually used as the foreign language is transparent and irrelevant for the System-Verilog side of this interface. Neither the SystemVerilog compiler nor the foreign language compiler is required to analyze the source code in the others language. Different programming languages can be used and supported with the same intact SystemVerilog layer.

DPI-C follows the principle of a black box: the specification and the implementation of a component are clearly separated, and the actual implementation is transparent to the rest of the system. Therefore, the actual programming language of the implementation is also transparent,

although this standard defines only C linkage semantics. The separation between SystemVerilog code and the foreign language is based on using functions as the natural encapsulation unit in SystemVerilog.

## **LAYERS**

### **Two Layers Of Dpi-C**

DPI-C consists of two separate layers: the SystemVerilog layer and a foreign language layer. The SystemVerilog layer does not depend on which programming language is actually used as the foreign language. Although different programming languages can be supported and used with the intact SystemVerilog layer, SystemVerilog defines a foreign language layer only for the C programming language. Nevertheless, SystemVerilog code shall look identical and its semantics shall be unchanged for any foreign language layer.

#### **Dpi-C Systemverilog Layer**

The SystemVerilog side of DPI-C does not depend on the foreign programming language. In particular, the actual function call protocol and argument passing mechanisms used in the foreign language are transparent and irrelevant to SystemVerilog. SystemVerilog code shall look identical regardless of what code the foreign side of the interface is using. The semantics of the SystemVerilog side of the interface is independent from the foreign side of the interface.

The SystemVerilog DPI-C allows direct inter-language function calls between SystemVerilog and any foreign programming language with a C function call protocol and linking model:

- ☞ Functions implemented in C and given import declarations in SystemVerilog can be called from SystemVerilog; such functions are referred to as imported functions.
- ☞ Functions implemented in SystemVerilog and specified in export declarations can be called from C; such functions are referred to as exported functions.
- ☞ Tasks implemented in SystemVerilog and specified in export declarations can be called from C; such functions are referred to as exported tasks.
- ☞ Functions implemented in C that can be called from SystemVerilog and can in turn call exported tasks; such functions are referred to as imported tasks.

#### **Dpi-C Foreign Language Layer**

The foreign language layer of the interface (which is transparent to SystemVerilog) shall specify how actual arguments are passed, how they can be accessed from the foreign code, how SystemVerilog-specific data types (such as logic and packed) are represented, and how they are translated to and from some predefined C-like types.

## **IMPORT**

### **Import Methods**

Methods implemented in C and given import declarations in SystemVerilog can be called from SystemVerilog, such methods are referred to as imported methods. Imported tasks or functions are similar to SystemVerilog tasks or functions. Imported tasks or functions can have zero or more formal input, output, and inout arguments.

Imported tasks always return an int result as part of the DPI-C disable protocol and, thus, are declared in foreign code as int functions. We will discuss about the DPI-C disable protocol in following sections.

Imported functions can return a result or be defined as void functions.

The syntax import method:

```
import {"DPI" | "DPI-C"} [context | pure] [c_identifier=] [function|task] [function_identifier|task_identifier] ([tf_port_list]);
```

### **Steps To Write Import Methods**

In SV Code

① Step1 : Import the C function

```
import "DPI-C" string_sv2c=task string_sv2c();
```

② Step2 : Invoke the Imported C function

```
initial
begin
    string_sv2c();
end
```

In C code:

③ Step3: Define the Imported function

```
void string_sv2c(){
    printf(" C: Hello from C ");
}
```

Full Example:

**CODE: SV\_file**

```

program main;
    string str;

    import "DPI-C" string_sv2c=task string_sv2c();

    initial
    begin
        string_sv2c();
    end

endprogram

```

**CODE: C\_file**

```

#include "svdpi.h"

```

```

void string_sv2c(){
    printf(" C: Hellow from C ");
}

```

## RESULTS

C: Hellow from C

**EXAMPLE: optional default arguments**

### Standard C Functions

Users can also call the standard C functions.

**EXAMPLE:**

```

import "DPI" function chandle malloc(int size);
import "DPI" function void free(chandle ptr);

```

### NAMING

#### Global Name

Every task or function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every task or function exported from SystemVerilog defines a global symbol. Global names of imported and exported tasks and functions must be unique (no overloading is allowed) and shall follow C conventions for naming; specifically, such names must start with a letter or underscore, and they can be followed by alphanumeric characters or underscores.

#### **EXAMPLE**

```
export "DPI-C" foo_plus = function \foo+; // "foo+" exported as "foo_plus"  
import "DPI-C" init_1 = function void \init[1] (); // "init_1" is a linkage name
```

#### **Local Name**

If a global name is not explicitly given, it shall be the same as the SystemVerilog task or function name.

#### **EXAMPLE:**

```
export "DPI-C" function foo;
```

#### **Sv Keyword As Linkage Name**

If a C method is named same as the SystemVerilog Keyword, then use a leading backslash ( \ ) character to create the linkage identifier.

#### **EXAMPLE:**

```
import "DPI-C" \begin = function void \init[2] (); // "begin" is a linkage name
```

#### **EXPORT**

#### **Export Methods**

Methods implemented in SystemVerilog and specified in export declarations can be called from C, such methods are referred to as exported methods.

#### **Steps To Write Export Methods**

In SV Code :

② Step1: Export the systemverilog function

```
export "DPI-C" function export_func;
```

② Step2: Define the systemverilog function

```
function void export_func();  
    $display("SV: Hello from SV ");  
endfunction
```

In C code :

⌚ Step3: Export the Systemverilog function

```
extern void export_func(void);
```

⌚ Step4: Invoke the systemverilog function

```
void import_func()
{
    export_func();
}
```

Full Example:

**CODE: SV\_file.sv**

```
program main;

export "DPI-C" function export_func;
import "DPI-C" function void import_func();

function void export_func();
    $display("SV: Hello from SV ");
endfunction

initial
begin
    import_func();
end

endprogram
```

**CODE: C\_file.c**

```
#include "stdio.h"
#include "vc_hdrs.h"
#include "svdpi.h"

extern void export_func(void);
```

```
void import_func()
{
    export_func();
}
```

## RESULTS:

SV: Hello from SV

### Blocking Export Dpi Task

SV Dpi allows C to call a SystemVerilog method which consumes time.

#### **CODE:SV\_file.sv**

```
program main;

export "DPI-C" task export_task;
import "DPI-C" context task import_task();

task export_task();
    $display("SV: Entered the export function . wait for some time : %0d ",$time);
    #100;
    $display("SV: After waiting %0d",$time);
endtask

initial
begin
    $display("SV: Before calling import function %0d",$time);
    import_task();
    $display("SV: After calling import function %0d",$time);
end
endprogram
```

**CODE: C\_file.c**

```
extern void export_task();

void import_task()
{
    printf(" C: Before calling export function\n");
    export_task();
    printf(" C: After calling export function\n");
}
```

## RESULTS

SV: Before calling import function 0  
C: Before calling export function  
SV: Entered the export function . wait for some time : 0  
SV: After waiting 100  
C: After calling export function  
SV: After calling import function 100

## PURE AND CONTEXT

### Pure Function

A function whose result depends solely on the values of its input arguments and with no side effects can be specified as pure. This can usually allow for more optimizations and thus can result in improved simulation performance.

A pure function call can be safely eliminated if its result is not needed or if the previous result for the same values of input arguments is available somehow and can be reused without needing to recalculate. Only nonvoid functions with no output or inout arguments can be specified as pure.

Specifically, a pure function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:

- ⌚ Perform any file operations.
- ⌚ Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.
- ⌚ Access any persistent data, like global or static variables.

### Context Function

Some DPI imported tasks or functions or other interface functions called from them require that the context of their call be known. The SystemVerilog context of DPI export tasks and functions must be known when they are called, including when they are called by imports. When an import invokes the svSetScope utility prior to calling the export, it sets the context explicitly. Otherwise, the context will be the context of the instantiated scope where the import declaration is located.

### **CODE: SV\_file\_1.sv**

**module** module\_1;

```
import "DPI-C" context function void import_func();  
export "DPI-C" function export_func;
```

```

module_2 instance_1();
initial
    import_func();

function void export_func();
    $display("SV: My scope is %m \n");
endfunction

endmodule

```

**CODE: SV\_file\_2.sv**

```

module module_2;

import "DPI-C" context function void import_func();
export "DPI-C" function export_func;

initial
    import_func();

function void export_func();
    $display("SV: My Scope is %m \n");
endfunction

endmodule

```

**CODE:C\_file.c**

```

#include "svdpi.h"
#include "stdio.h"

extern void export_func(void);

void import_func()
{
    printf(" C: Im called fromm Scope :: %s \n\n ",svGetNameFromScope(svGetScope()));
    export_func();
}

```

## RESULTS

C: Im called fromm Scope :: module\_1.instance\_1

SV: My Scope is module\_1.instance\_1.export\_func

C: Im called fromm Scope :: module\_1

SV: My scope is module\_1.export\_func

## **DATA TYPES**

The SystemVerilog DPI supports only SystemVerilog data types, which are the data types that can cross the boundary between SystemVerilog and a foreign language in both the direction. On the other hand, the data types used in C code shall be C types. A value that is passed through the DPI is specified in SystemVerilog code as a value of SystemVerilog data type, while the same value is declared C code as a value of C data type. Therefore, a pair of matching type definitions is required to pass a value through DPI, the SystemVerilog definition and the C definition.

The following SystemVerilog types are the only permitted types for formal arguments of import and export tasks or functions:

- ⌚ void, byte, shortint, int, longint, real, shortreal, chandle, and string
- ⌚ Scalar values of type bit and logic
- ⌚ Packed arrays, structs, and unions composed of types bit and logic. Every packed type is eventually equivalent to a packed one-dimensional array. On the foreign language side of the DPI, all packed types are perceived as packed one-dimensional arrays regardless of their declaration in the SystemVerilog code.
- ⌚ Enumeration types interpreted as the type associated with that enumeration
- ⌚ Types constructed from the supported types with the help of the constructs: struct , union , Unpacked array , typedef

Mapping data types

SystemVerilog type	C type
byte	char
shortint	short int
Int	int
Longint	long long
Real	double
Shortreal	float
chandle	void*
string	const char*
Bit	unsigned char
logica/reg	unsigned char

© www.testbench.in

### Passing Logic Datatype

The DPI defines the canonical representation of packed 2-state (type svBitVecVal) and 4-state arrays (type svLogicVecVal). svLogicVecVal is fully equivalent to type s\_vpi\_vecval, which is used to represent 4-state logic in VPI.

4-state Value Map

SV	C Data	C control
0	0	0
1	1	0
x	0	1
z	1	1

© www.testbench.in

### CODE:SV\_file.sv

```
program main;
logic a;
import "DPI" function void show(logic a);
initial begin
  a = 1'b0;
  show(a);
end
```

```

a = 1'b1;
show(a);
a = 1'bX;
show(a);
a = 1'bZ;
show(a);
end
endprogram

```

#### **CODE: C\_file.v**

```

#include <stdio.h>
#include <svdpi.h>

void show(svLogic a){
    if(a == 0)
        printf(" a is 0 \n");
    else if(a == 1)
        printf(" a is 1 \n");
    else if(a == 2)
        printf(" a is x \n");
    else if(a == 3)
        printf(" a is z \n");
}

```

#### **RESULTS**

a is 0  
a is 1  
a is z  
a is x

### **ARRAYS**

#### **Open Arrays**

The size of the packed dimension, the unpacked dimension, or both dimensions can remain unspecified, such cases are referred to as open arrays (or unsized arrays). Open arrays allow the use of generic code to handle different sizes. Formal arguments in SystemVerilog can be specified as open arrays solely in import declarations, exported. SystemVerilog functions cannot have formal arguments specified as open arrays.

OpenArrays are good for generic programming, since C language doesn't have concept of

parameterizable arguments. Standard query and library functions are provided to determine array information to access array elements.

#### EXAMPLE: open arrays

##### CODE:SV\_file.sv

```
program main;

int fxd_arr_1[8:3];
int fxd_arr_2[12:1];

import "DPI-C" context function void pass_array(input int dyn_arr[] );

initial
begin
  for (int i = 3; i<=8 ; i++)
    begin
      fxd_arr_1[i] = $random();
      $display("SV:fxd_arr_1 %0d %d ",i, fxd_arr_1[i]);
    end

  $display("\n Passing fxd_arr_1 to C \n");
  pass_array( fxd_arr_1 );

  for (int i = 1; i<=12 ; i++)
    begin
      fxd_arr_2[i] = $random();
      $display("SV: fxd_arr_2 %0d %d ",i, fxd_arr_2[i]);
    end

  $display("\n Passing fxd_arr_2 to C \n");
  pass_array( fxd_arr_2 );
end
endprogram
```

##### CODE: C\_file.c

```
#include <stdio.h>
#include <svdpi.h>
```

```

void pass_array(const svOpenArrayHandle dyn_arr ) {
    int i;

    printf("Array Left %d, Array Right %d \n\n", svLeft(dyn_arr,1), svRight(dyn_arr, 1));
    for (i= svRight(dyn_arr,1); i <= svLeft(dyn_arr,1); i++) {
        printf("C: %d %d \n", i, *(int*)svGetArrElemPtr1(dyn_arr, i));
    }
    printf("\n\n");
}

```

### RESULTS:

SV:fxd\_arr\_1 3 303379748  
 SV:fxd\_arr\_1 4 -1064739199  
 SV:fxd\_arr\_1 5 -2071669239  
 SV:fxd\_arr\_1 6 -1309649309  
 SV:fxd\_arr\_1 7 112818957  
 SV:fxd\_arr\_1 8 1189058957

Passing fxd\_arr\_1 to C

Array Left 8, Array Right 3

C: 3 303379748  
 C: 4 -1064739199  
 C: 5 -2071669239  
 C: 6 -1309649309  
 C: 7 112818957  
 C: 8 1189058957

SV: fxd\_arr\_2 1 -1295874971  
 SV: fxd\_arr\_2 2 -1992863214  
 SV: fxd\_arr\_2 3 15983361  
 SV: fxd\_arr\_2 4 114806029  
 SV: fxd\_arr\_2 5 992211318  
 SV: fxd\_arr\_2 6 512609597  
 SV: fxd\_arr\_2 7 1993627629  
 SV: fxd\_arr\_2 8 1177417612  
 SV: fxd\_arr\_2 9 2097015289  
 SV: fxd\_arr\_2 10 -482925370  
 SV: fxd\_arr\_2 11 -487095099

SV: fxd\_arr\_2 12 -720121174

Passing fxd\_arr\_2 to C

Array Left 12, Array Right 1

C: 1 -1295874971  
C: 2 -1992863214  
C: 3 15983361  
C: 4 114806029  
C: 5 992211318  
C: 6 512609597  
C: 7 1993627629  
C: 8 1177417612  
C: 9 2097015289  
C: 10 -482925370  
C: 11 -487095099  
C: 12 -720121174

### Packed Arrays

A packed array is represented as an array of one or more elements (of type svBitVecVal for 2-state values and svLogicVecVal for 4-state values), each element representing a group of 32 bits.

**CODE:SV\_file.sv**

```
program main;  
  
import "DPI-C" function void get_nums(output logic [15:0] nums[10]);  
  
logic [15:0] nums[10];  
  
initial begin  
    get_nums(nums);  
    foreach (nums[i]) $display(i,nums[i]);  
end  
endprogram
```

**CODE:C\_file.c**

```
#include "svdpi.h"  
  
void fib(svLogicVecVal nums[10]) {
```

```

int i;
for (i=0; i<10; i++) {
    nums[i] = i ;
}
}

```

## RESULTS:

```

0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9

```

### Linearized And Normalized

Arrays **use** normalized ranges **for** the **packed** [n-1:0] **and** the unpacked part [0:n-1]

For example, **if SV** code defines an array as follows:

```
logic [2:3][1:3][2:0] b [1:10][31:0];
```

Then C code would see it as defined like **this**:

```
logic [17:0] b [0:9][0:31];
```

### Array Querying Functions

- ⌚ svLeft() shall return the left bound (MSB) of the dimension.
- ⌚ svRight() shall return the right bound (LSB) of the dimension.
- ⌚ svLow() shall return the minimum of left index and right index of the dimension.
- ⌚ svHigh() shall return the maximum of left index and right index of the dimension.

- svIncrement() shall return 1 if left index is greater than or equal to right index and -1 if left index is less than right index.
- svLength() shall return the number of elements in the dimension, which is equivalent to high index - low index + 1.
- svDimensions() shall return total number of dimensions in the array

#### CODE: SV\_file.sv

```

program main;
  int fxd_arr_1[8:3];
  int fxd_arr_2[1:13];
import "DPI-C" context function void pass_array(input int dyn_arr[] );
initial
begin
  $display("\n Passing fxd_arr_1 to C \n");
  pass_array( fxd_arr_1 );
  $display("\n Passing fxd_arr_2 to C \n");
  pass_array( fxd_arr_2 );
end
endprogram

```

#### CODE: C\_file.c

```

#include <stdio.h>
#include <svdpi.h>
void pass_array(const svOpenArrayHandle dyn_arr ) {
  printf("Array Pointer is %x \n", svGetArrayPtr(dyn_arr) );
  printf(" Lower index %d \n", svLow(dyn_arr,1));
  printf(" Higher index %d \n", svHigh(dyn_arr, 1) );
  printf(" Left index %d \n", svLeft(dyn_arr,1), svRight(dyn_arr, 1) );
  printf(" Right index %d \n", svRight(dyn_arr, 1) );
  printf(" Length of array %d \n", svLength(dyn_arr,1) );
  printf(" Incremental %d \n", svIncrement(dyn_arr,1));
  printf(" Dimentions of Array %d \n", svDimensions(dyn_arr));
  printf(" Size of Array in bytes %d \n", svSizeOfArray(dyn_arr) );
}

```

#### RESULTS:

Passing fxd\_arr\_1 to C

Array Pointer is 80fdc58  
Lower index 3  
Higher index 8  
Left index 8  
Right index 3  
Length of array 6  
Incremental 1  
Dimentions of Array 1  
Size of Array in bytes 24

Passing fxd\_arr\_2 to C

Array Pointer is 80fdc70  
Lower index 1  
Higher index 13  
Left index 1  
Right index 13  
Length of array 13  
Incremental -1  
Dimentions of Array 1  
Size of Array in bytes 52

## **PASSING STRUCTS AND UNIONS**

### **Passing Structure Example**

DPI allows to pass the structs and Unions . This can be done by passing pointers or by packing.

In the following example, a "struct" is passed from SystemVerilog to C and also from C to Systemverilog using import and export functions. While passing the "struct" data type, the data is packed in to array and passed from SV to C and then the array is decoded back to Struct in C. The same when the Struct is passed from C to SystemVerilog.

#### **CODE: C\_file.c**

```
#include "stdio.h"  
#include "vc_hdrs.h"  
#include "svdpi.h"  
extern "C" {  
    typedef struct {  
        int a;  
        int b;  
        char c;
```

```

} C_struct;
extern void export_func(svBitVec32 x[3] );
void import_func()
{
    C_struct s_data;
    unsigned int arr[3];

    s_data.a = 51;
    s_data.b = 242;
    s_data.c = 35;
    printf( "C : s_data.a = %d\n", s_data.a );
    printf( "C : s_data.b = %d\n", s_data.b );
    printf( "C : s_data.c = %d\n\n", s_data.c );

    arr[0] = s_data.a ;
    arr[1] = s_data.b ;
    arr[2] = s_data.c ;

    export_func(arr);
}

}

```

#### CODE: SV\_file.sv

```

program main;
    export "DPI-C" function export_func;
    import "DPI-C" function void import_func();

typedef struct packed{
    int a;
    int b;
    byte c;
} SV_struct;

function void export_func(input int arr[3]);

SV_struct s_data;

s_data.a = arr[0];
s_data.b = arr[1]; s_data.c = arr[2];

```

```

$display("SV: s_data.a = %0d", s_data.a );
$display("SV: s_data.b = %0d", s_data.b );
$display("SV: s_data.c = %0d \n", s_data.c );
endfunction
initial
begin
    import_func();
end
endprogram

```

#### RESULTS:

```

C : s_data.a = 51
C : s_data.b = 242
C : s_data.c = 35

```

```

SV: s_data.a = 51
SV: s_data.b = 242
SV: s_data.c = 35

```

#### Passing Openarray Structs

CODE: C\_file.c

```

#include "svdpi.h"

typedef struct {int p; int q} PkdStru;

void send2c(const svOpenArrayHandle dyn_arr)
{
    int i;
    PkdStru Sele;
    printf("\n\n Array Left %d, Array Right %d \n\n", svLeft(dyn_arr,1), svRight(dyn_arr, 1) );
    for (i= svLeft(dyn_arr,1); i <= svRight(dyn_arr,1); i++) {
        Sele = *(PkdStru*)svGetArrElemPtr1(dyn_arr, i);
        printf("C : %d : [%d,%d]\n",i, Sele.q,Sele.p );
    }
    printf("\n\n");
}

```

CODE: SV\_file.sv

**program** open\_array\_struct ();

```

typedef struct packed { int p; int q; } PkdStru;

```

```

import "DPI-C" function void send2c (input PkdStru arr []);
PkdStru arr_data [0:4];
initial begin
foreach (arr_data[i]) begin
    arr_data[i] = {$random,$random};
    $display("SV: %0d : [%0d,%0d]",i,arr_data[i].p,arr_data[i].q);
end
    send2c(arr_data);
end
endprogram

```

#### **RESULTS:**

```

SV: 0 : [303379748,-1064739199]
SV: 1 : [-2071669239,-1309649309]
SV: 2 : [112818957,1189058957]
SV: 3 : [-1295874971,-1992863214]
SV: 4 : [15983361,114806029]
Array Left 0, Array Right 4

```

```

C : 0 : [303379748,-1064739199]
C : 1 : [-2071669239,-1309649309]
C : 2 : [112818957,1189058957]
C : 3 : [-1295874971,-1992863214]
C : 4 : [15983361,114806029]

```

#### **Passing Union Example**

**CODE:SV\_file**

```

module m;
typedef bit [2:0] A;

typedef union packed { A a; S s; } U;
U u;
A a;
// Import function takes three arguments
import "DPI-C" function void foo8(input A fa, input U fu);
initial begin
    a = 3'b100;
    u.a = 3'b100;
    foo8(a, u);
end

```

```

endmodule
CODE:C_file
#include "svdpi.h"
void foo8(
const svBitVecVal* fa,
const svBitVecVal* fu)
{
    printf("fa is %d, fu is %d\n", *fa, *fu);
}

```

### **ARGUMENTS TYPE**

#### **What You Specify Is What You Get**

For input and inout arguments, the temporary variable is initialized with the value of the actual argument with the appropriate coercion. For output or inout arguments, the value of the temporary variable is assigned to the actual argument with the appropriate conversion.

Arguments specified in SystemVerilog as input must not be modified by the foreign language code. The initial values of formal arguments specified in SystemVerilog as output are undetermined and implementation dependent.

#### **Pass By Ref**

For arguments passed by reference, a reference (a pointer) to the actual data object is passed. In the case of packed data, a reference to a canonical data object is passed. The actual argument is usually allocated by a caller. The caller can also pass a reference to an object already allocated somewhere else, for example, its own formal argument passed by reference. If an argument of type T is passed by reference, the formal argument shall be of type T\*. Packed arrays are passed using a pointer to the appropriate canonical type definition, either svLogicVecVal\* or svBitVecVal\*.

#### **Pass By Value**

Only small values of formal input arguments are passed by value. Function results are also directly passed by value. The user needs to provide the C type equivalent to the SystemVerilog type of a formal argument if an argument is passed by value.

#### **Passing String**

The layout of SystemVerilog string objects is implementation dependent. However, when a string value is passed from SystemVerilog to C, implementations shall ensure that all characters in the string are laid out in memory per C string conventions, including a trailing null character present at the end of the C string.

#### **Example : Passing String From Sv To C**

##### **CODE: SV\_file.sv**

```

program main;
    string str;

```

```

import "DPI-C" string_sv2c=task string_sv2c(string str);
initial
begin
    str = " HELLO: This string is created in SystemVerilog \n" ;
    string_sv2c(str);
end
endprogram
CODE: C_file.c
#include "svdpi.h"
int string_sv2c(const char* str){

    printf(" C: %s",str);
    return 0;

}

```

## RESULTS

C: HELLO: This string is created in SystemVerilog

### Example: Passing String From C To Sv

From the Data type mapping table, a SystemVerilog "String" is mapped to "const char\*" in C. In the Following example, string "HELLO: This string is created in C" is assigned to a string and passed as return value to function import "string\_c2sv" and this import function is called in SystemVerilog.

#### CODE: SV\_file.v

```

program main;
    string str;
    import "DPI-C" context function string string_c2sv();

initial
begin
    str = string_c2sv();
    $display(" SV: %s ",str);
end
endprogram

```

#### CODE: C\_file.c

```

#include "svdpi.h"
const char* string_c2sv(void) {
    char* str;
    str = " HELLO: This string is created in C ";
    return str;
}

```

}

## RESULTS:

SV: HELLO: This string is created in C

## **DISABLIE**

### **Disable Dpi-C Tasks And Functions**

It is possible for a disable statement to disable a block that is currently executing a mixed language call chain. When a DPI import task or function is disabled, the C code is required to follow a simple disable protocol. The protocol gives the C code the opportunity to perform any necessary resource cleanup, such as closing open file handles, closing open VPI handles, or freeing heap memory.

The protocol is composed of the following items:

- ⌚ a) When an exported task returns due to a disable, it must return a value of 1. Otherwise, it must return 0.
- ⌚ b) When an imported task returns due to a disable, it must return a value of 1. Otherwise, it must return 0.
- ⌚ c) Before an imported function returns due to a disable, it must call the API function svAckDisabledState().
- ⌚ d) Once an imported task or function enters the disabled state, it is illegal for the current function invocation to make any further calls to exported tasks or functions.

## **Include Files**

Applications that use the DPI with C code usually need this main include file. The include file svdpi.h defines the types for canonical representation of 2-state (bit) and 4-state (logic) values and passing references to SystemVerilog data objects. The file also provides function headers and defines a number of helper macros and constants. The content of svdpi.h does not depend on any particular implementation; all simulators shall use the same file.

## **INTRODUCTION**

the UVM (Universal Verification Methodology) Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog.

UVM library contains:

- ⌚ Component classes for building testbench components like generator/driver/monitor etc.
- ⌚ Reporting classes for logging,
- ⌚ Factory for object substitution.
- ⌚ Synchronization classes for managing concurrent process.
- ⌚ Policy classes for printing, comparing, recording, packing, and unpacking of uvm\_object based classes.
- ⌚ TLM Classes for transaction level interface.
- ⌚ Sequencer and Sequence classes for generating realistic stimulus.
- ⌚ And Macros which can be used for shorthand notation of complex implementation. In this

tutorial, we will learn some of the UVM concepts with examples.

### **Installing Uvm Library**

- 1)Go to <http://www.accellera.org/activities/vip/>
- 2)Download the uvm\*.tar.gz file.
- 3)Untar the file.
- 4)Go to the extracted directory : cd uvm\*\uvm\src
- 5)Set the UVM\_HOME path : setenv UVM\_HOME `pwd`  
(This is required to run the examples which are downloaded from this site)
- 6)Go to examples : cd ..\examples\hello\_world\uvm/
- 7)Compile the example using :  
your\_tool\_compilation\_command -f compile\_<toolname>.f  
(example for questasim use : qverilog -f compile\_questa.f)
- 8)Run the example.

### **UVM TESTBENCH**

Uvm components, uvm env and uvm test are the three main building blocks of a testbench in uvm based verification.

#### **Uvm env**

uvm\_env is extended from uvm\_component and does not contain any extra functionality.  
uvm\_env is used to create and connect the uvm\_components like driver, monitors , sequencers etc. A environment class can also be used as sub-environment in another environment. As there is no difference between uvm\_env and uvm\_component , we will discuss about uvm\_component, in the next section.

#### **Verification Components**

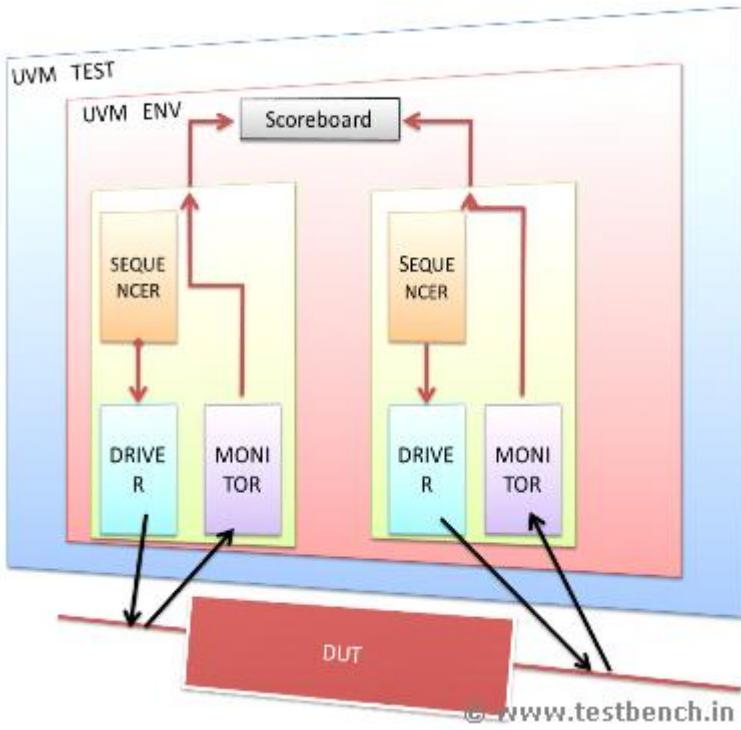
uvm verification component classes are derived from uvm\_component class which provides features like hierarchy searching, phasing, configuration , reporting , factory and transaction recording.

Following are some of the uvm component classes

- uvm\_agent
- uvm\_monitor
- uvm\_scoreboard
- uvm\_driver
- uvm\_sequencer

NOTE: uvm\_env and uvm\_test are also extended from uvm\_component.

A typical uvm verification environment:



An agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

#### **About Uvm component Class:**

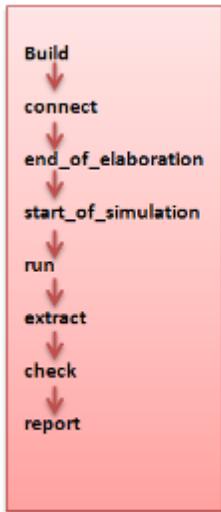
uvm\_compoenent class is inherited from uvm\_report\_object which is inherited from uvm\_object.

As I mentioned previously, uvm\_component class provides features like hierarchy searching, phasing, configuration , reporting , factory and transaction recording.

We will discuss about phasing concept in this section and rest of the features will be discussed as separate topics.

#### **UVM phases**

UVM Components execute their behavior in strictly ordered, pre-defined phases. Each phase is defined by its own virtual method, which derived components can override to incorporate component-specific behavior. By default , these methods do nothing.



© www.bsoftnch.in

### --> **virtual function void** build()

This phase is used to construct various child components/ports/exports and configures them.

### --> **virtual function void** connect()

This phase is used for connecting the ports/exports of the components.

### --> **virtual function void** end\_of\_elaboration()

This phase is used for configuring the components if required.

### --> **virtual function void** start\_of\_simulation()

This phase is used to print the banners and topology.

### --> **virtual task** run()

In this phase , Main body of the test is executed where all threads are forked off.

### --> **virtual function void** extract()

In this phase, all the required information is gathered.

### --> **virtual function void** check()

In this phase, check the results of the extracted information such as un responded requests in

scoreboard, read statistics registers etc.

--> **virtual function void** report()

This phase is used for reporting the pass/fail status.

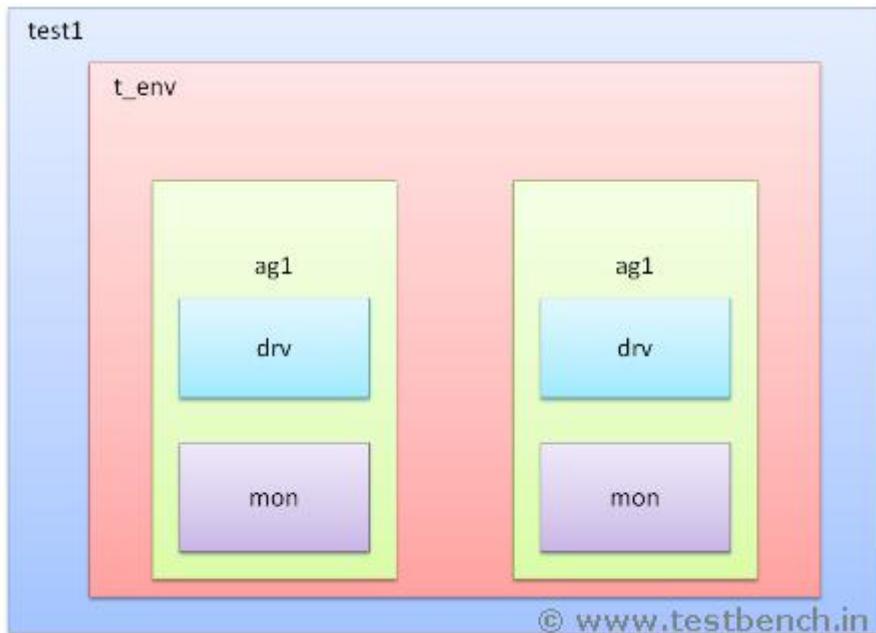
Only build() method is executed in top down manner. i.e after executing parent build() method, child objects build() methods are executed. All other methods are executed in bottom-up manner. The run() method is the only method which is time consuming. The run() method is forked, so the order in which all components run() method are executed is undefined.

### Uvm test

uvm\_test is derived from uvm\_component class and there is no extra functionality is added. The advantage of used uvm\_test for defining the user defined test is that the test case selection can be done from command line option +UVM\_TESTNAME=<testcase\_string>. User can also select the testcase by passing the testcase name as string to uvm\_root::run\_test(<testcase\_string>) method.

In the above <testcase\_string> is the object type of the testcase class.

Lets implement environment for the following topology. I will describe the implementation of environment , testcase and top module. Agent, monitor and driver are implemented similar to environment.



1)Extend uvm\_env class and define user environment.

```
class env extends uvm_env;
```

2)Declare the utility macro. This utility macro provides the implementation of create() and get\_type\_name() methods and all the requirements needed for factory.

```
`uvm_component_utils(env)
```

3)Declare the objects for agents.

```
agent ag1;  
agent ag2;
```

4)Define the constructor. In the constructor, call the super methods and pass the parent object. Parent is the object in which environment is instantiated.

```
function new(string name , uvm_component parent = null);  
    super.new(name, parent);  
endfunction: new
```

5)Define build method. In the build method, construct the agents.

To construct agents, use create() method. The advantage of create() over new() is that when create() method is called, it will check if there is a factory override and constructs the object of override type.

```
function void build();  
    super.build();  
    uvm_report_info(get_full_name(),"Build", UVM_LOW);  
    ag1 = agent::type_id::create("ag1",this);  
    ag2 = agent::type_id::create("ag2",this);  
endfunction
```

6)Define connect(),end\_of\_elaboration(),start\_of\_simulation(),run(),extract(),check(),report() methods.

Just print a message from these methods, as we dont have any logic in this example to define.

```
function void connect();  
    uvm_report_info(get_full_name(),"Connect", UVM_LOW);  
endfunction
```

Complete code of environment class:

```
class env extends uvm_env;  
  
`uvm_component_utils(env)  
agent ag1;
```

```

agent ag2;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  function void build();
    uvm_report_info(get_full_name(),"Build", UVM_LOW);
    ag1 = agent::type_id::create("ag1",this);
    ag2 = agent::type_id::create("ag2",this);
  endfunction
  function void connect();
    uvm_report_info(get_full_name(),"Connect", UVM_LOW);
  endfunction
  function void end_of_elaboration();
    uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
  endfunction
  function void start_of_simulation();
    uvm_report_info(get_full_name(),"Start_of_simulation", UVM_LOW);
  endfunction
  task run();
    uvm_report_info(get_full_name(),"Run", UVM_LOW);
  endtask
  function void extract();
    uvm_report_info(get_full_name(),"Extract", UVM_LOW);
  endfunction
  function void check();
    uvm_report_info(get_full_name(),"Check", UVM_LOW);
  endfunction
  function void report();
    uvm_report_info(get_full_name(),"Report", UVM_LOW);
  endfunction
endclass

```

Now we will implement the testcase.

1)Extend uvm\_test and define the test case

```
class test1 extends uvm_test;
```

2)Declare component utilts using utility macro.

```
`uvm_component_utils(test1)
```

2)Declare environment class handle.

```
env t_env;
```

3)Define constructor method. In the constructor, call the super method and construct the environment object.

```

function new (string name="test1", uvm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new

```

4) Define the end\_of\_elaboration method. In this method, call the print() method. This print() method will print the topology of the test.

```

function void end_of_elaboration();
    uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
    print();
endfunction

```

4) Define the run method and call the global\_stop\_request() method.

```

task run ();
    #1000;
    global_stop_request();
endtask : run

```

#### Testcase source code:

```

class test1 extends uvm_test;
`uvm_component_utils(test1)
env t_env;
function new (string name="test1", uvm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new
function void end_of_elaboration();
    uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
    print();
endfunction
task run ();
    #1000;
    global_stop_request();
endtask : run

```

**endclass**

#### Top Module:

To start the testbench, run\_test() method must be called from initial block.

Run\_test() method phases all components through all registered phases.

```

module top;
initial
    run_test();

```

**endmodule**

## Download the source code

[uvm\\_phases.tar](#)

[Browse the code in uvm\\_phases.tar](#)

## Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

### Log file:

```
[RNTST] Running test test1...
uvm_test_top.t_env [uvm_test_top.t_env] Build
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Build
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Build
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Build
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Build
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Build
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Build
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Connect
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Connect
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Connect
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Connect
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Connect
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Connect
uvm_test_top.t_env [uvm_test_top.t_env] Connect
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] End_of_elaboration
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] End_of_elaboration
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] End_of_elaboration
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] End_of_elaboration
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] End_of_elaboration
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] End_of_elaboration
uvm_test_top.t_env [uvm_test_top.t_env] End_of_elaboration
uvm_test_top [uvm_test_top] End_of_elaboration
```

---

Name	Type	Size	Value
<hr/>			
uvm_test_top	test1	-	uvm_test_top@2
t_env	env	-	t_env@4
ag1	agent	-	ag1@6
drv	driver	-	drv@12
rsp_port	uvm_analysis_port	-	rsp_port@16

```

sqr_pull_port  uvm_seq_item_pull_+ -      sqr_pull_port@14
mon           monitor       -          mon@10
ag2           agent         -          ag2@8
drv           driver        -          drv@20
rsp_port      uvm_analysis_port -      rsp_port@24
sqr_pull_port  uvm_seq_item_pull_+ -      sqr_pull_port@22
mon           monitor       -          mon@18

```

```

uvm_test_top.t_env.ag1.drv[uvm_test_top.t_env.ag1.drv]Start_of_simulation
uvm_test_top.t_env.ag1.mon[uvm_test_top.t_env.ag1.mon]Start_of_simulation
uvm_test_top.t_env.ag1[uvm_test_top.t_env.ag1]Start_of_simulation
uvm_test_top.t_env.ag2.drv[uvm_test_top.t_env.ag2.drv]Start_of_simulation
uvm_test_top.t_env.ag2.mon[uvm_test_top.t_env.ag2.mon]Start_of_simulatio

```

```

..
..
..
..
```

Observe the above log report:

1)Build method was called in top-down fashion. Look at the following part of message.

```

uvm_test_top.t_env [uvm_test_top.t_env] Build
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Build
uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Build
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Build
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Build
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Build
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Build

```

2)Connect method was called in bottom up fashion. Look at the below part of log file,

```

uvm_test_top.t_env.ag1.drv [uvm_test_top.t_env.ag1.drv] Connect
uvm_test_top.t_env.ag1.mon [uvm_test_top.t_env.ag1.mon] Connect
uvm_test_top.t_env.ag1 [uvm_test_top.t_env.ag1] Connect
uvm_test_top.t_env.ag2.drv [uvm_test_top.t_env.ag2.drv] Connect
uvm_test_top.t_env.ag2.mon [uvm_test_top.t_env.ag2.mon] Connect
uvm_test_top.t_env.ag2 [uvm_test_top.t_env.ag2] Connect
uvm_test_top.t_env [uvm_test_top.t_env] Connect

```

3)Following part of log file shows the testcase topology.

Name	Type	Size	Value
uvm_test_top	test1	-	uvm_test_top@2
t_env	env	-	t_env@4
ag1	agent	-	ag1@6
drv	driver	-	drv@12
rsp_port	uvm_analysis_port	-	rsp_port@16
sqr_pull_port	uvm_seq_item_pull_*	-	sqr_pull_port@14
mon	monitor	-	mon@10
ag2	agent	-	ag2@8
drv	driver	-	drv@20
rsp_port	uvm_analysis_port	-	rsp_port@24
sqr_pull_port	uvm_seq_item_pull_*	-	sqr_pull_port@22
mon	monitor	-	mon@12

© www.testbencha.in

## UVM REPORTING

The uvm\_report\_object provides an interface to the UVM reporting facility. Through this interface, components issue the various messages with different severity levels that occur during simulation. Users can configure what actions are taken and what file(s) are output for individual messages from a particular component or for all messages from all components in the environment.

A report consists of an id string, severity, verbosity level, and the textual message itself. If the verbosity level of a report is greater than the configured maximum verbosity level of its report object, it is ignored.

### Reporting Methods:

Following are the primary reporting methods in the UVM.

**virtual function void** uvm\_report\_info  
**(string id,string message,int verbosity=UVM\_MEDIUM,string filename="",int line=0)**

**virtual function void** uvm\_report\_warning  
**(string id,string message,int verbosity=UVM\_MEDIUM,string filename="",int line=0)**

**virtual function void** uvm\_report\_error  
**(string id,string message,int verbosity=UVM\_LOW, string filename="",int line=0)**

**virtual function void** uvm\_report\_fatal  
**(string id,string message,int verbosity=UVM\_NONE, string filename="",int line=0)**

### Arguments description:

© id -- a unique id to form a group of messages.

© message -- The message text

⌚ verbosity -- the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level, then the report is issued, subject to the configured action and file descriptor settings.

⌚ filename/line -- If required to print filename and line number from where the message is issued, use macros, `\_\_FILE\_\_` and `\_\_LINE\_\_`.

### **Actions:**

These methods associate the specified action or actions with reports of the given severity, id, or severity-id pair.

Following are the actions defined:

- ⌚ UVM\_NO\_ACTION -- Do nothing
- ⌚ UVM\_DISPLAY -- Display report to standard output
- ⌚ UVM\_LOG -- Write to a file
- ⌚ UVM\_COUNT -- Count up to a max\_quit\_count value before exiting
- ⌚ UVM\_EXIT -- Terminates simulation immediately
- ⌚ UVM\_CALL\_HOOK -- Callback the hook method .

### **Configuration:**

Using these methods, user can set the verbosity levels and set actions.

```
function void set_report_verbosity_level  
    (int verbosity_level)  
function void set_report_severity_action  
    (uvm_severity severity,uvm_action action)  
function void set_report_id_action  
    (string id,uvm_action action)  
function void set_report_severity_id_action  
    (uvm_severity severity,string id,uvm_action action)
```

UVM Reporting API		
	Types	Methods
<b>Severity</b>	UVM_INFO	uvm_report_info
	UVM_WARNING	uvm_report_warning
	UVM_ERROR	uvm_report_error
	UVM_FATAL	uvm_report_fatal
<b>Filtering</b>	set_report_verbosity_level	
<b>Actions</b>	Types	Methods
	UVM_NO_ACTION	set_report_id_action
	UVM_DISPLAY	
	UVM_LOG	set_report_severity_action
	UVM_COUNT	
	UVM_EXIT	set_report_severity_id_action
<b>Outputfile</b>	set_report_default_file	
	set_report_severity_file	
	set_report_id_file	
	set_report_severity_id_file	
<b>Query</b>	get_report_verbosity_level	
	get_report_action	
	get_report_file_handle	

© www.testbench.in

### Example

Lets see an example:

In the following example, messages from rpting::run() method are of different verbosity level. In the top module, 3 objects of rpting are created and different verbosity levels are set using set\_report\_verbosity\_level() method.

```

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

class rpting extends uvm_component;

`uvm_component_utils(rpting)
function new(string name,uvm_component parent);
    super.new(name, parent);
endfunction

task run();
    uvm_report_info(get_full_name(),
        "Info Message : Verbo lvl - UVM_NONE ",UVM_NONE,`__FILE__,`__LINE__);
    uvm_report_info(get_full_name(),
        "Info Message : Verbo lvl - UVM_LOW ",UVM_LOW);
endtask

```

```

uvm_report_info(get_full_name(),
 "Info Message : Verbo lvl - 150      ",150);
uvm_report_info(get_full_name(),
 "Info Message : Verbo lvl - UVM_MEDIUM",UVM_MEDIUM);

uvm_report_warning(get_full_name(),
 "Warning Messgae from rpting",UVM_LOW);

uvm_report_error(get_full_name(),
 "Error Message from rpting \n\n",UVM_LOW);
endtask
endclass
module top;
rpting rpt1;
rpting rpt2;
rpting rpt3;
initial begin
rpt1 = new("rpt1",null);
rpt2 = new("rpt2",null);
rpt3 = new("rpt3",null);
rpt1.set_report_verbosity_level(UVM_MEDIUM);
rpt2.set_report_verbosity_level(UVM_LOW);
rpt3.set_report_verbosity_level(UVM_NONE);
run_test();
end
endmodule

```

[Download the source code](#)

[uvm\\_reporting.tar](#)

[Browse the code in uvm\\_reporting.tar](#)

### Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

### Log file:

```

UVM_INFO reporting.sv(13)@0:rpt1[rpt1]Info Message:Verbo lvl - UVM_NONE
UVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - UVM_LOW
UVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - 150
UVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - UVM_MEDIUM
UVM_WARNIN@0:rpt[rpt1] Warning Messgae from rpting
UVM_ERROR @0:rpt1[rpt1] Error Message from rpting

```

```

UVM_INFOreporting.sv(13)@0:rpt2[rpt2]Info Message:Verbo lvl - UVM_NONE
UVM_INFO @ 0:rpt2[rpt2] Info Message : Verbo lvl - UVM_LOW
UVM_WARNING@0:rpt2[rpt2] Warning Messgae from rpting
UVM_ERROR@0:rpt2[rpt2] Error Message from rpting
UVM_INFOreporting.sv(13)@0:rpt3[rpt3]Info Message:Verbo lvl - UVM_NONE
UVM_ERROR @ 9200 [TIMOUT] Watchdog timeout of '23f0' expired.

```

### **UVM TRANSACTION**

A transaction is data item which is eventually or directly processed by the DUT. The packets, instructions, pixels are data items. In uvm, transactions are extended from uvm\_transactions class or uvm\_sequence\_item class. uvm\_transaction is a typedef of uvm\_sequence\_item.

#### **Example of a transaction:**

```

class Packet extends uvm_transaction;
    rand bit [7:0] da;
    rand bit [7:0] sa;
    rand bit [7:0] length;
    rand bit [7:0] data[];
    rand byte fcs;
endclass

```

#### **Core Utilities:**

uvm\_transaction class is extended from uvm\_object. uvm\_transaction adds more features like transaction recording , transaction id and timings of the transaction.

The methods used to model, transform or operate on transactions like print, copying, cloning, comparing, recording, packing and unpacking are already defined in uvm\_object.

Utilities Method	Description
print	Prints this object's properties
record	Records this object's properties
copy	Returns a deep copy of this object.
compare	Deep compares this data object with the object provided in the rhs (right-hand side) argument.
pack	This method bitwise-concatenates this object's properties into an array
unpack	Extracts property values from an array
clone	Clone method creates and returns an exact copy of this object. The default implementation calls create method followed by copy method.
create	This method allocates a new object of the same type as this object and returns it via a base uvm_object handle.

## OBJECT UTILITIES

### User Defined Implementations:

User should define these methods in the transaction using do\_<method\_name> and call them using <method\_name>. Following table shows calling methods and user-defined hook do\_<method\_name> methods. Clone and create methods, does not use hook methods concepts.

Functionality	User defined hook methods	Calling methods
Printing	do_print	print
		sprint
		convert2string
Recording	do_record	record
Copying	do_copy	copy
Comparing	do_compare	compare
Packing	do_pack	pack
		pack_bytes
		pack_ints
Unpacking	do_unpack	unpack
		unpack_bytes
		unpack_ints
Cloning	clone	clone
Create	create	create

© www.testbench.in

### Shorthand Macros:

Using the field automation concept of uvm, all the above defines methods can be defined automatically.

To use these field automation macros, first declare all the data fields, then place the field automation macros between the `uvm\_object\_utils\_begin and `uvm\_object\_utils\_end macros.

### Example of field automation macros:

```
class Packet extends uvm_transaction;  
    rand bit [7:0] da;  
    rand bit [7:0] sa;  
    rand bit [7:0] length;  
    rand bit [7:0] data[];  
    rand byte    fcs;
```

```

`uvm_object_utils_begin(Packet)
  `uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_int(sa, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_int(length, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_array_int(data, UVM_ALL_ON|UVM_NOPACK)
  `uvm_field_int(fcs, UVM_ALL_ON|UVM_NOPACK)
`uvm_object_utils_end
endclass.

```

For most of the data types in systemverilog, uvm defined corresponding field automation macros. Following table shows all the field automation macros.

Type	Macros
Scalar	`uvm_field_int `uvm_field_object `uvm_field_string `uvm_field_enum `uvm_field_real `uvm_field_event
Static Array	`uvm_field_sarray_int `uvm_field_sarray_object `uvm_field_sarray_string `uvm_field_sarray_enum
Dynamic array	`uvm_field_array_int `uvm_field_array_object `uvm_field_array_string `uvm_field_array_enum
Queue	`uvm_field_queue_int `uvm_field_queue_object `uvm_field_queue_string `uvm_field_queue_enum
Associative array with string index	`uvm_field_aa_int_string `uvm_field_aa_object_string `uvm_field_aa_string_string
Associative array with integral index	`uvm_field_aa_object_int `uvm_field_aa_int_int `uvm_field_aa_int_int_unsigned `uvm_field_aa_int_integer `uvm_field_aa_int_integer_unsigned `uvm_field_aa_int_byte `uvm_field_aa_int_byte_unsigned `uvm_field_aa_int_shortint `uvm_field_aa_int_shortint_unsigned `uvm_field_aa_int_longint `uvm_field_aa_int_longint_unsigned `uvm_field_aa_int_key `uvm_field_aa_int_enumkey

Each `uvm\_field\_\*` macro has at least two arguments: ARG and FLAG.

ARG is the instance name of the variable and FLAG is used to control the field usage in core utilities operation.

Following table shows uvm field automation flags:

FLAG	DESCRIPTION
UVM_ALL_ON	Set all operations
UVM_DEFAULT	Use the default flag settings.
UVM_NOCOPY	Do not copy this field.
UVM_NOCOMPARE	Do not compare this field.
UVM_NOPRINT	Do not print this field.
UVM_NODEFPRINT	Do not print the field if it is the same as its
UVM_NOPACK	Do not pack or unpack this field.
UVM_PHYSICAL	Treat as a physical field.
UVM_ABSTRACT	Treat as an abstract field.
UVM_READONLY	Do not allow setting of this field from the set_* local methods.

© www.testbench.in

By default, FLAG is set to UVM\_ALL\_ON. All these flags can be ored. Using NO\_`\*` flags, can turn off particular field usage in a paerticular method. NO\_`\*` flags takes precedence over other flags.

### Example of Flags:

```
`uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
```

The above macro will use the field "da" in all utilities methods except Packing and unpacking methods.

### Lets see a example:

In the following example, all the utility methods are defined using field automation macros except Packing and unpacking methods. Packing and unpacking methods are done in do\_pack() and do\_unpack() method.

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

//Define the enumerated types for packet types
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;
class Packet extends uvm_transaction;
  rand fcs_kind_t  fcs_kind;
  rand bit [7:0] length;
  rand bit [7:0] da;
```

```

rand bit [7:0] sa;
rand bit [7:0] data[];
rand byte fcs;
constraint payload_size_c { data.size inside { [1 : 6];} }
constraint length_c { length == data.size; }
function new(string name = "");
    super.new(name);
endfunction : new
function void post_randomize();
    if(fcs_kind == GOOD_FCS)
        fcs = 8'b0;
    else
        fcs = 8'b1;
    fcs = cal_fcs();
endfunction : post_randomize
//// method to calculate the fcs ////
virtual function byte cal_fcs;
    integer i;
    byte result ;
    result = 0;
    result = result ^ da;
    result = result ^ sa;
    result = result ^ length;
    for (i = 0;i < data.size;i++)
        result = result ^ data[i];
    result = fcs ^ result;
    return result;
endfunction : cal_fcs
`uvm_object_utils_begin(Packet)
`uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(sa, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(length, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_array_int(data, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(fcs, UVM_ALL_ON|UVM_NOPACK)
`uvm_object_utils_end
function void do_pack(uvm_packer packer);
    super.do_pack(packer);
    packer.pack_field_int(da,$bits(da));
    packer.pack_field_int(sa,$bits(sa));
    packer.pack_field_int(length,$bits(length));

```

```

foreach(data[i])
    packer.pack_field_int(data[i],8);
    packer.pack_field_int(fcs,$bits(fcs));
endfunction : do_pack
function void do_unpack(uvm_packer packer);
    int sz;
    super.do_pack(packer);

    da = packer.unpack_field_int($bits(da));
    sa = packer.unpack_field_int($bits(sa));
    length = packer.unpack_field_int($bits(length));

    data.delete();
    data = new[length];
    foreach(data[i])
        data[i] = packer.unpack_field_int(8);
    fcs = packer.unpack_field_int($bits(fcs));
endfunction : do_unpack

endclass : Packet

```

```

///////////
/// Test to check the packet implementation ///
///////////

module test;
    Packet pkt1 = new("pkt1");
    Packet pkt2 = new("pkt2");
    byte unsigned pkdbytes[];
initial
    repeat(10)
        if(pkt1.randomize)
            begin
                $display(" Randomization Sucessesfull.");
                pkt1.print();
                uvm_default_packer.use_metadata = 1;
                void`(pkt1.pack_bytes(pkdbytes));
                $display("Size of pkd bits %d",pkdbytes.size());
                pkt2.unpack_bytes(pkdbytes);
                pkt2.print();

```

```

if(pkt2.compare(pkt1))
    $display(" Packing,Unpacking and compare worked");
else
    $display(" *** Something went wrong in Packing or Unpacking or compare *** \n \n");
end
else
    $display(" *** Randomization Failed ***");
endmodule

```

Download the source code

[uvm\\_transaction.tar](#)

[Browse the code in uvm\\_transaction.tar](#)

Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

Log report:

Randomization Sucessesfull.			
Name	Type	Size	Value
pkt1	Packet	-	pkt1@3
da	integral	8	'h1d
sa	integral	8	'h26
length	integral	8	'h5
data	da(integral)	5	-
[0]	integral	8	'hb1
[1]	integral	8	'h3f
[2]	integral	8	'h9e
[3]	integral	8	'h38
[4]	integral	8	'h8d
fcs	integral	8	'h9b
Size of pkd bits		9	
Name	Type	Size	Value
pkt2	Packet	-	pkt2@5
da	integral	8	'h1d
sa	integral	8	'h26

length	integral	8	'h5
data	da(integral)	5	-
[0]	integral	8	'hb1
[1]	integral	8	'h3f
[2]	integral	8	'h9e
[3]	integral	8	'h38
[4]	integral	8	'h8d
fcs	integral	8	'h9b

Packing,Unpacking and compare worked

## **UVM CONFIGURATION**

Configuration is a mechanism in UVM that higher level components in a hierarchy can configure the lower level components variables. Using set\_config\_\* methods, user can configure integer, string and objects of lower level components. Without this mechanism, user should access the lower level component using hierarchy paths, which restricts reusability. This mechanism can be used only with components. Sequences and transactions cannot be configured using this mechanism. When set\_config\_\* method is called, the data is stored w.r.t strings in a table. There is also a global configuration table.

Higher level component can set the configuration data in level component table. It is the responsibility of the lower level component to get the data from the component table and update the appropriate table.

### **Set config \* Methods:**

Following are the method to configure integer , string and object of uvm\_object based class respectively.

```
function void set_config_int (string inst_name,
                           string field_name,
                           uvm_bitstream_t value)
function void set_config_string (string inst_name,
                                string field_name,
                                string value)
function void set_config_object (string inst_name,
                                string field_name,
                                uvm_object value, bit clone = 1)
```

### **Arguments description:**

- ⌚ **string** inst\_name: Hierarchical string path.
- ⌚ **string** field\_name: Name of the field in the table.
- ⌚ **bitstream\_t** value: In set\_config\_int, a integral value that can be anything from 1 bit to 4096 bits.
- ⌚ **bit** clone : If this bit is set then object is cloned.

inst\_name and field\_name are strings of hierachal path. They can include wile card "\*" and "?" characters. These methods must be called in build phase of the component.

"\*" matches zero or more characters

"?" matches exactly one character

### **Some examples:**

"\*" -All the lower level components.

"\*abc" -All the lower level components which ends with "abc".

Example: "xabc","xyabc","xyzabc" ....

"abc\*" -All the lower level components which starts with "abc".

Example: "abcx","abcxy","abcxyz" ....

"ab?" -All the lower level components which start with "ab" , then followed by one more character.

Example: "abc","abb","abx" ....

"?bc" -All the lower level components which start with any one character ,then followed by "c".

Example: "abc","xbc","bbc" ....

"a?c" -All the lower level components which start with "a" , then followed by one more character and then followed by "c".

Example: "abc","aac","axc" ..

There are two ways to get the configuration data:

1)Automatic : Using Field macros

2)Manual : using gte\_config\_\* methods.

### **Automatic Configuration:**

To use the atomic configuration, all the configurable fields should be defined using uvm component field macros and uvm component utilities macros.

#### **uvm component utility macros:**

For non parameterized classes

`uvm\_component\_utils\_begin(TYPE)

`uvm\_field\_\* macro invocations here

`uvm\_component\_utils\_end

For parameterized classes.

`uvm\_component\_param\_utils\_begin(TYPE)

`uvm\_field\_\* macro invocations her

```
`uvm_component_utils_end
```

For UVM Field macros, Refer to link

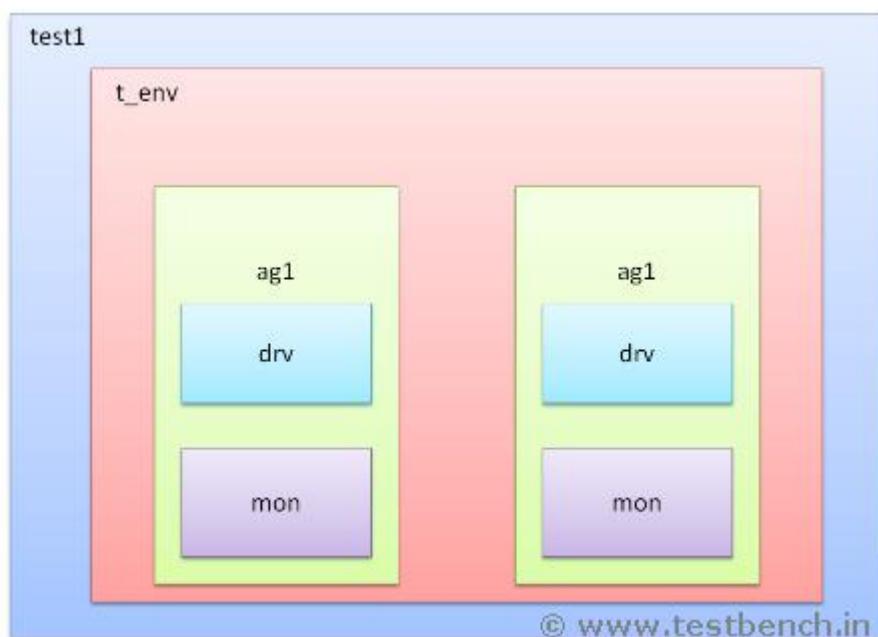
### [UVM TRANSACTION](#)

Example:

Following example is from link

### [UVM TESTBENCH](#)

2 Configurable fields, a integer and a string are defined in env, agent, monitor and driver classes.  
Topology of the environment using these classes is



### Driver class Source Code:

Similar to driver class, all other components env, agent and monitor are define.

```
class driver extends uvm_driver;
  integer int_cfg;
  string str_cfg;
`uvm_component_utils_begin(driver)
  `uvm_field_int(int_cfg, UVM_DEFAULT)
  `uvm_field_string(str_cfg, UVM_DEFAULT)
`uvm_component_utils_end
function new(string name, uvm_component parent);
  super.new(name, parent);
endfunction
function void build();
  super.build();

```

**endfunction**

**endclass**

## Testcase:

Using `set_config_int()` and `set_config_string()` configure variables at various hierachal locations.

```
//t_env.ag1.drv.int_cfg
//t_env.ag1.mon.int_cfg
set_config_int("* .ag1.*", "int_cfg", 32);
//t_env.ag2.drv
set_config_int("t_env.ag2.drv", "int_cfg", 32);
//t_env.ag2.mon
set_config_int("t_env.ag2.mon", "int_cfg", 32);
//t_env.ag1.mon.str_cfg
//t_env.ag2.mon.str_cfg
//t_env.ag1.drv.str_cfg
//t_env.ag2.drv.str_cfg
set_config_string("* .ag?.*", "str_cfg", "pars");
//t_env.str_cfg
set config string("t_env", "str_cfg", "abcd");
```

## Download the source code

## uvm configuration 1.tar

Browse the code in uvm\_configuration\_1.tar

### **Command to run the simulation**

---

VCS Users : make vcs

## Questa Users: make questa

From the above log report of the example, we can see the variables int\_cfg and str\_cfg of all the components and they are as per the configuration setting from the testcase.

## Manual Configurations:

Using `get_config_*` methods, user can get the required data if the data is available in the table. Following are the method to get configure data of type integer , string and object of `uvm_object` based class respectively.

If a entry is found in the table with "field\_name" then data will be updated to "value" argument . If entry is not found, then the function returns "0". So when these methods are called, check the return value.

Example:

**Driver class code:**

```
class driver extends uvm_driver;
    integer int_cfg;
    string str_cfg;

    `uvm_component_utils(driver)
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    function void build();
        super.build();
        void'(get_config_int("int_cfg",int_cfg));
        void'(get_config_string("str_cfg",str_cfg));
        uvm_report_info(get_full_name(),
            $psprintf("int_cfg %0d : str_cfg %0s ",int_cfg,str_cfg),UVM_LOW);
    endfunction
endclass
```

**Download the source code**

[uvm\\_configuration\\_2.tar](#)

[Browse the code in uvm\\_configuration\\_2.tar](#)

**Command to run the simulation**

VCS Users : make vcs

Questa Users: make questa

**Log file**

```
UVM_INFO @ 0: uvm_test_top.t_env
    int_cfg x : str_cfg abcd
UVM_INFO @ 0: uvm_test_top.t_env.ag1
    int_cfg x : str_cfg
UVM_INFO @ 0: uvm_test_top.t_env.ag1.drv
    int_cfg 32 : str_cfg pars
UVM_INFO @ 0: uvm_test_top.t_env.ag1.mon
    int_cfg 32 : str_cfg pars
UVM_INFO @ 0: uvm_test_top.t_env.ag2
    int_cfg x : str_cfg
UVM_INFO @ 0: uvm_test_top.t_env.ag2.drv
    int_cfg 32 : str_cfg pars
```

UVM\_INFO @ 0: uvm\_test\_top.t\_env.ag2.mon

int\_cfg 32 : str\_cfg pars

### **Configuration Setting Members:**

#### **print\_config\_settings**

```
function void print_config_settings  
  (string field = "",  
   uvm_component comp = null,  
   bit recurse = 0)
```

This method prints all configuration information for this component.

If "field" is specified and non-empty, then only configuration settings matching that field, if any, are printed. The field may not contain wildcards. If "recurse" is set, then information for all children components are printed recursively.

#### **print\_config\_matches**

```
static bit print_config_matches = 0
```

Setting this static variable causes get\_config\_\* to print info about matching configuration settings as they are being applied. These two members will be helpful to know while debugging.

### **Download the source code**

#### **uvm\_configuration\_3.tar**

#### **Browse the code in uvm\_configuration\_3.tar**

#### **Command to run the simulation**

VCS Users : make vcs

Questa Users: make questa

#### **Log file**

When print\_config\_settings method is called

uvm\_test\_top.t\_env.ag1.drv

  uvm\_test\_top.\*.ag1.\* int\_cfg int 32

  uvm\_test\_top.t\_env.ag1.drv.rsp\_port

    uvm\_test\_top.\*.ag?.\* str\_cfg string pars

  uvm\_test\_top.t\_env.ag1.drv.rsp\_port

    uvm\_test\_top.\*.ag1.\* int\_cfg int 32

  uvm\_test\_top.t\_env.ag1.drv.sqr\_pull\_port

    uvm\_test\_top.\*.ag?.\* str\_cfg string pars

  uvm\_test\_top.t\_env.ag1.drv.sqr\_pull\_port

    uvm\_test\_top.\*.ag1.\* int\_cfg int 32

When print\_config\_matches is set to 1.

UVM\_INFO @ 0: uvm\_test\_top.t\_env [auto-configuration]

Auto-configuration matches for component uvm\_test\_top.t\_env (env).

Last entry for a given field takes precedence.

Config set from	Instance Path	Field name	Type	Value
-----------------	---------------	------------	------	-------

---

```
uvm_test_top(test1) uvm_test_top.t_env str_cfg string abcd
```

## **UVM FACTORY**

The factory pattern is an well known object-oriented design pattern. The factory method design pattern defining a separate method for creating the objects. , whose subclasses can then override to specify the derived type of object that will be created.

Using this method, objects are constructed dynamically based on the specification type of the object. User can alter the behavior of the pre-build code without modifying the code. From the testcase, user from environment or testcase can replace any object which is at any hierarchy level with the user defined object.

For example: In your environment, you have a driver component. You would like the extend the driver component for error injection scenario. After defining the extended driver class with error injection, how will you replace the base driver component which is deep in the hierarchy of your environment ? Using hierarchical path, you could replace the driver object with the extended driver. This could not be easy if there are many driver objects. Then you should also take care of its connections with the other components of testbenchs like scoreboard etc.

One more example: In your Ethernet verification environment, you have different drivers to support different interfaces for 10mbps,100mps and 1G. Now you want to reuse the same environment for 10G verification. Inside somewhere deep in the hierarchy, while building the components, as a driver components ,your current environment can only select 10mmmps/100mps/1G drivers using configuration settings. How to add one more driver to the current drivers list of drivers so that from the testcase you could configure the environment to work for 10G.

Using the uvm fatroy, it is very easy to solve the above two requirements. Only classss extended from uvm\_object and uvm\_component are supported for this.

There are three basic steps to be followed for using uvm factory.

- ⌚ 1) Registration
- ⌚ 2) Construction
- ⌚ 3) Overriding

The factory makes it is possible to override the type of uvm component /object or instance of a uvm component/object in2 ways. They are based on uvm component/object type or uvm componenent/object name.

### **Registration:**

While defining a class , its type has to be registered with the uvm factory. To do this job easier, uvm has predefined macros.

```
'uvm_component_utils(class_type_name)
'uvm_component_param_utils(class_type_name #(params))
'uvm_object_utils(class_type_name)
'uvm_object_param_utils(class_type_name #(params))
```

For uvm\_\*\_param\_utils are used for parameterized classes and other two macros for non-parameterized class. Registration is required for name-based overriding , it is not required for type-based overriding.

**EXAMPLE: Example of above macros**

```
class packet extends uvm_object;
  `uvm_object_utils(packet)
endclass

class packet #(type T=int, int mode=0) extends uvm_object;
  `uvm_object_param_utils(packet #(T,mode))
endclass

class driver extends uvm_component;
  `uvm_component_utils(driver)
endclass

class monitor #(type T=int, int mode=0) extends uvm_component;
  `uvm_component_param_utils(driver#(T,mode))
endclass
```

**Construction:**

To construct a uvm based component or uvm based objects, static method create() should be used. This function constructs the appropriate object based on the overrides and constructs the object and returns it. So while constructing the uvm based components or uvm based objects , do not use new() constructor.

Syntax :

```
static function T create(string name,
                        uvm_component parent,
                        string context = " ")
```

The Create() function returns an instance of the component type, T, represented by this proxy, subject to any factory overrides based on the context provided by the parents full name. The context argument, if supplied, supersedes the parents context. The new instance will have the given leaf name and parent.

**EXAMPLE:**

```
class_type object_name;
object_name = clss_type::type_id::creat("object_name",this);
```

For uvm\_object based classes, doesnt need the parent handle as second argument.

**Overriding:**

If required, user could override the registered classes or objects. User can override based of name string or class-type.

There are 4 methods defined for overriding:

```
function void set_inst_override_by_type  
  (uvm_object_wrapper original_type,  
   uvm_object_wrapper override_type,  
   string full_inst_path )
```

The above method is used to override the object instances of "original\_type" with "override\_type" . "override\_type" is extended from "original\_type".

```
function void set_inst_override_by_name  
  (string original_type_name,  
   string override_type_name,  
   string full_inst_path )
```

Original\_type\_name and override\_type\_name are the class names which are registered in the factory. All the instances of objects with name "Original\_type\_name" will be overriden with objects of name "override\_type\_name" using set\_inst\_override\_by\_name() method.

```
function void set_type_override_by_type  
  (uvm_object_wrapper original_type,  
   uvm_object_wrapper override_type,  
   bit replace = 1 )
```

Using the above method, request to create an object of original\_type can be overriden with override\_type.

```
function void set_type_override_by_name  
  (string original_type_name,  
   string override_type_name,  
   bit replace = 1 )
```

Using the above method, request to create an object of original\_type\_name can be overriden with override\_type\_name.

When multiple overrides are done , then using the argument "replace" , we can control whether to override the previous override or not. If argument "replace" is 1, then previous overrides will be replaced otherwise, previous overrides will remain.

print() method, prints the state of the uvm\_factory, registered types, instance overrides, and type overrides.

Now we will see a complete example. This example is based on the environment build in topic UVM TESTBENCH . Refer to that section for more information about this example.

Lets look at the 3 steps which I discussed above using the example defined in UVM TESTBENCH

### ①) Registration

In all the class, you can see the macro `uvm\_component\_utils(type\_name)

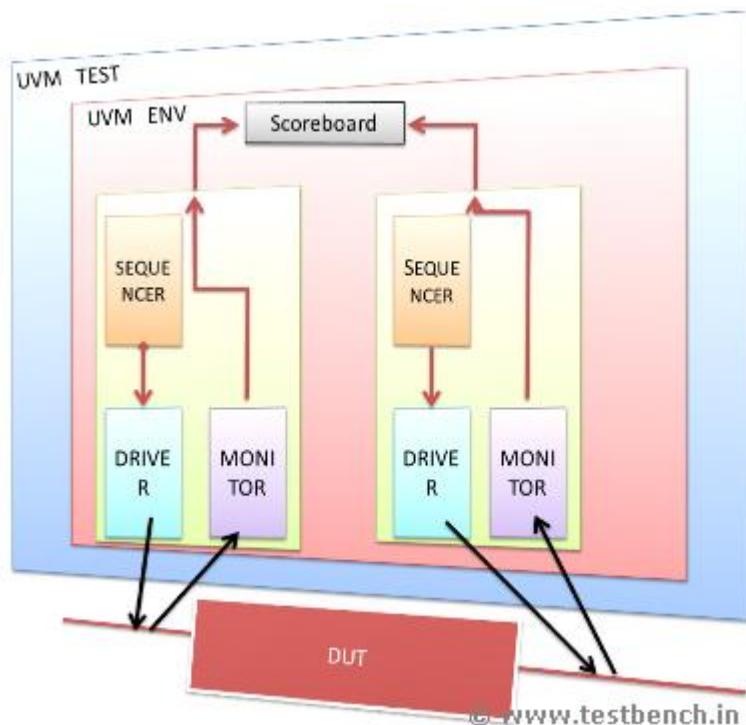
### ②) Construction

In file agant.sv file, monitor and driver are constructed using create() method.

```
mon = monitor::type_id::create("mon",this);  
drv = driver::type_id::create("drv",this);
```

④ 3)In this example, a one testcase is already developed in topic UVM\_TESTBENCH. There are no over rides in this test case.

Topology of this test environment is shown below.



In this example, there is one driver class and one monitor class. In this testcase , By extending driver class , we will define driver\_2 class and by extending monitor class, we will define monitor\_2 class.

From the testcase , Using set\_type\_override\_by\_type, we will override driver with driver\_2 and Using set\_type\_override\_by\_name, we will override monitor with monitor\_2.

To know about the overrides which are done, call factory.print() method of factory class.

```

class driver_2 extends driver;
  `uvm_component_utils(driver_2)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

endclass

class monitor_2 extends monitor;
  `uvm_component_utils(monitor_2)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass
```

```

class test_factory extends uvm_test;
`uvm_component_utils(test_factory)
  env t_env;
  function new (string name="test1", uvm_component parent=null);
    super.new (name, parent);
    factory.set_type_override_by_type(driver::get_type(),driver_2::get_type(),"*");
    factory.set_type_override_by_name("monitor","monitor_2","");
    factory.print();
    t_env = new("t_env",this);
  endfunction : new
  function void end_of_elaboration();
    uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
    print();
  endfunction : end_of_elaboration
  task run ();
    #1000;
    global_stop_request();
  endtask : run
endclass

```

[Download the example:](#)

[uvm\\_factory.tar](#)

[Browse the code in uvm\\_factory.tar](#)

[Command to simulate](#)

Command to run the example with the testcase which is defined above:

VCS Users : make vcs

Questa Users: make questa

Method factory.print() displayed all the overrides as shown below in the log file.

##### Factory Configuration (\*)

No instance overrides are registered with this factory

Type Overrides:

Requested Type    Override Type

Requested Type	Override Type
driver	driver_2
monitor	monitor_2

In the below text printed by print\_topology() method ,we can see overridden driver and monitor.

Name	Type	Size	Value
uvm_test_top	test_factory	-	uvm_test_top@2
t_env	env	-	t_env@4
ag1	agent	-	ag1@6
drv	driver_2	-	drv@12
rsp_port	uvm_analysis_port	-	rsp_port@16
sqr_pull_port	uvm_seq_item_pull_+ -	-	sqr_pull_port@14
mon	monitor_2	-	mon@10
ag2	agent	-	ag2@8
drv	driver_2	-	drv@20
rsp_port	uvm_analysis_port	-	rsp_port@24
sqr_pull_port	uvm_seq_item_pull_+ -	-	sqr_pull_port@22
mon	monitor_2	-	mon@18

In the below text printed by print\_topology() method ,with testcase test1 which does note have overrides.

Command to run this example with test1 is

VCS Users : make vcs

Questa Users: make questa

Name	Type	Size	Value
uvm_test_top	test1	-	uvm_test_top@2
t_env	env	-	t_env@4
ag1	agent	-	ag1@6
drv	driver	-	drv@12
rsp_port	uvm_analysis_port	-	rsp_port@16
sqr_pull_port	uvm_seq_item_pull_+ -	-	sqr_pull_port@14
mon	monitor	-	mon@10
ag2	agent	-	ag2@8
drv	driver	-	drv@20

```

rsp_port      uvm_analysis_port -      rsp_port@24
  sqr_pull_port  uvm_seq_item_pull_+ -  sqr_pull_port@22
  mon          monitor      -          mon@18

```

## **UVM SEQUENCE 1**

### **Introduction**

A sequence is a series of transaction. User can define the complex stimulus. sequences can be reused, extended, randomized, and combined sequentially and hierarchically in various ways.

For example, for a processor, lets say PUSH\_A,PUSH\_B,ADD,SUB,MUL,DIV and POP\_C are the instructions. If the instructions are generated randomly, then to excusing a meaningful operation like "adding 2 variables" which requires a series of transaction

"PUSH\_A PUSH\_B ADD POP\_C " will take longer time. By defining these series of "PUSH\_A PUSH\_B ADD POP\_C ", it would be easy to exercise the DUT.

Advantages of uvm sequences :

- ⌚ Sequences can be reused.
- ⌚ Stimulus generation is independent of testbench.
- ⌚ Easy to control the generation of transaction.
- ⌚ Sequences can be combined sequentially and hierarchically.

A complete sequence generation requires following 4 classes.

- 1- Sequence item.
- 2- Sequence
- 3- Sequencer
- 4- Driver

- ⌚ uvm\_sequence\_item :

User has to define a transaction by extending uvm\_sequence\_item. uvm\_sequence\_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism. For more information about uvm\_sequence\_item Refer to link

### **UVM TRANSACTION**

- ⌚ uvm\_sequence:

User should extend uvm\_sequence class and define the construction of sequence of transactions. These transactions can be directed, constrained randomized or fully randomized. The uvm\_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

```

virtual class uvm_sequence #(  

    type REQ = uvm_sequence_item,  

    type RSP = REQ  

)

```

**© uvm\_sequencer:**

uvm\_sequencer is responsible for the coordination between sequence and driver. Sequencer sends the transaction to driver and gets the response from the driver. The response transaction from the driver is optional. When multiple sequences are running in parallel, then sequencer is responsible for arbitrating between the parallel sequences. There are two types of sequencers : uvm\_sequencer and uvm\_push\_sequencer

```

class uvm_sequencer #(  

    type REQ = uvm_sequence_item,  

    type RSP = REQ  

)  

class uvm_push_sequencer #(  

    type REQ = uvm_sequence_item,  

    type RSP = REQ  

)

```

**© uvm driver:**

User should extend uvm\_driver class to define driver component. uvm driver is a component that initiate requests for new transactions and drives it to lower level components. There are two types of drivers: uvm\_driver and uvm\_push\_driver.

```

class uvm_driver #(  

    type REQ = uvm_sequence_item,  

    type RSP = REQ  

)  

class uvm_push_driver #(  

    type REQ = uvm_sequence_item,  

    type RSP = REQ  

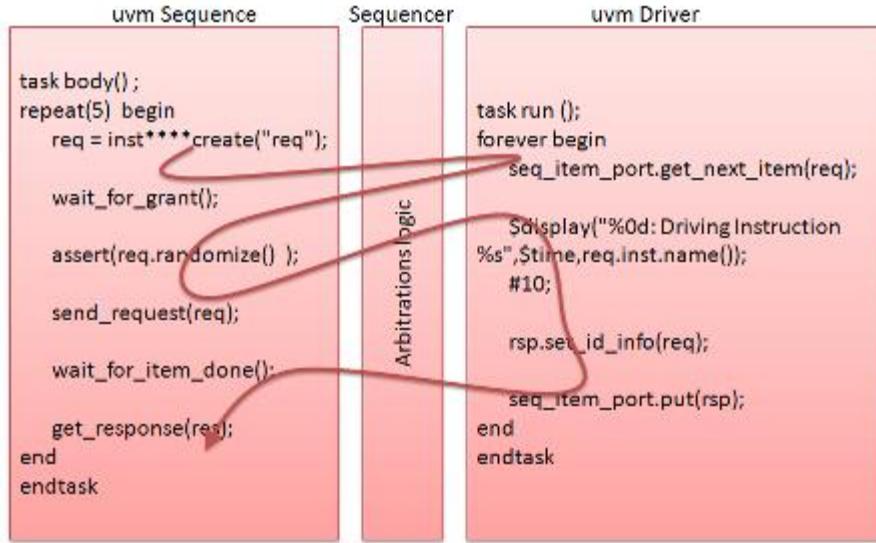
)

```

In pull mode , uvm\_sequencer is connected to uvm\_driver , in push mode uvm\_push\_sequencer is connectd to uvm\_push\_driver.

uvm\_sequencer and uvm\_driver are parameterized components with request and response transaction types. REQ and RSP types by default are uvm\_sequence\_type types. User can specify REQ and RSP of different transaction types. If user specifies only REQ type, then RSP will be REQ type.

## Sequence And Driver Communication:



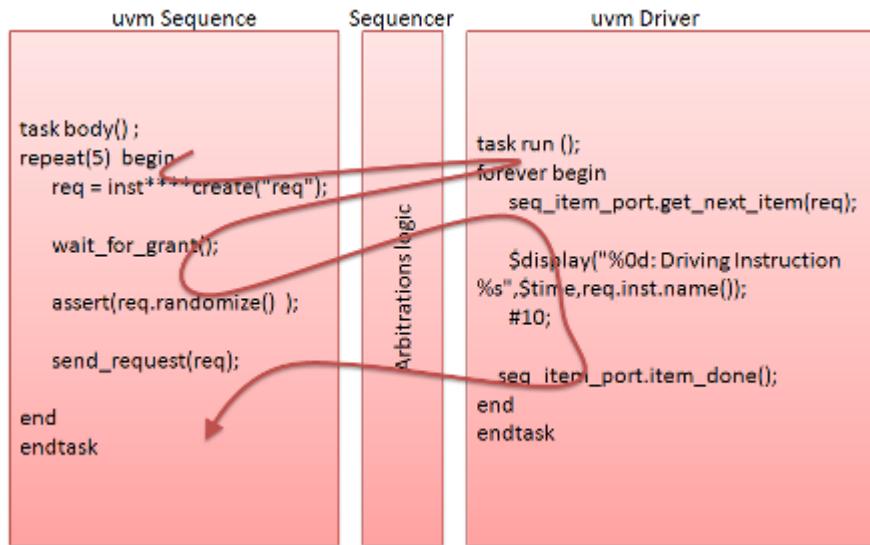
The above image shows how a transaction from a sequence is sent to driver and the response from the driver is sent to sequencer. There are multiple methods called during this operation.

First when the body() method is called

- 1) A transaction is created using "create()" method. If a transaction is created using "create()" method, then it can be overridden if required using uvm factory.
- 2) After a transaction is created, wait\_for\_grant() method is called. This method is blocking method.
- 3) In the run task of the driver, when "seq\_item\_port.get\_next\_item()" is called, then the sequencer un blocks wait\_for\_grant() method. If more than one sequence is getting executed by sequencer, then based on arbitration rules, un blocks the wait\_for\_grant() method.
- 4) After the wait\_for\_grant() un blocks, then transaction can be randomized, or its properties can be filled directly. Then using the send\_request() method, send the transaction to the driver.
- 5) After calling the send\_request() method, "wait\_for\_item\_done()" method is called. This is a blocking method and execution gets blocks at this method call.
- 6) The transaction which is sent from sequence , in the driver this transaction is available as "seq\_item\_port.get\_next\_item(req)" method argument. Then driver can drive this transaction to bus or lower level.

7) Once the driver operations are completed, then by calling "seq\_item\_port.put(rsp)", wait\_for\_item\_done() method of sequence gest unblocked. Using get\_responce(res), the response transaction from driver is taken by sequence and processes it.

After this step, again the steps 1 to 7 are repeated five times.



If a response from driver is not required, then steps 5,6,7 can be skipped and item\_done() method from driver should be called as shown in above image.

### Simple Example

Let write an example: This is a simple example of processor instruction. Various instructions which are supported by the processor are PUSH\_A,PUSH\_B,ADD,SUB,MUL,DIV and POP\_C.

#### Sequence Item

1) Extend uvm\_sequence\_item and define instruction class.

```
class instruction extends uvm_sequence_item;
```

2) Define the instruction as enumerated types and declare a variable of instruction enumerated type.

```
typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
rand inst_t inst;
```

3) Define operational method using uvm\_field\_\* macros.

```
`uvm_object_utils_begin(instruction)
`uvm_field_enum(inst_t,inst, UVM_ALL_ON)
`uvm_object_utils_end
```

4) Define the constructor.

```
function new (string name = "instruction");
    super.new(name);
endfunction
```

### Sequence item code:

```
class instruction extends uvm_sequence_item;
  typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
  rand inst_t inst;
  `uvm_object_utils_begin(instruction)
    `uvm_field_enum(inst_t,inst, UVM_ALL_ON)
  `uvm_object_utils_end
  function new (string name = "instruction");
    super.new(name);
  endfunction
endclass
```

### Sequence

We will define a operation addition using uvm\_sequence. The instruction sequence should be "PUSH A PUSH B ADD POP C".

1) Define a sequence by extending uvm\_sequence. Set REQ parameter to "instruction" type.

```
class operation_addition extends uvm_sequence #(instruction);
```

2) Define the constructor.

```
function new(string name="operation_addition");
  super.new(name);
endfunction
```

3) Lets name the sequencer which we will develop is "instruction\_sequencer".

Using the `uvm\_sequence\_utils macro, register the "operation\_addition" sequence with "instruction\_sequencer" sequencer. This macro adds the sequence to the sequencer list. This macro will also register the sequence for factory overrides.

```
`uvm_sequence_utils(operation_addition, instruction_sequencer)
```

4)

In the body() method, first call wait\_for\_grant(), then construct a transaction and set the instruction enum to PUSH\_A . Then send the transaction to driver using send\_request() method. Then call the wait\_for\_item\_done() method. Repeat the above steps for other instructions PUSH\_B, ADD and POP\_C.

For construction of a transaction, we will use the create() method.

```
virtual task body();
  req = instruction::type_id::create("req");
  wait_for_grant();
  assert(req.randomize() with {
    inst == instruction::PUSH_A;
  });
  send_request(req);
  wait_for_item_done();
```

```

//get_response(res); This is optional. Not using in this example.
req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::PUSH_B;
send_request(req);
wait_for_item_done();
//get_response(res);
req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::ADD;
send_request(req);
wait_for_item_done();
//get_response(res);

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::POP_C;
send_request(req);
wait_for_item_done();
//get_response(res);
endtask

```

### Sequence code

```

class operation_addition extends uvm_sequence #(instruction);
instruction req;
function new(string name="operation_addition");
  super.new(name);
endfunction
`uvm_sequence_utils(operation_addition, instruction_sequencer)
virtual task body();
  req = instruction::type_id::create("req");
  wait_for_grant();
  assert(req.randomize() with {
    inst == instruction::PUSH_A;
  });
  send_request(req);
  wait_for_item_done();
//get_response(res); This is optional. Not using in this example.

req = instruction::type_id::create("req");
wait_for_grant();

```

```

req.inst = instruction::PUSH_B;
send_request(req);
wait_for_item_done();
//get_response(res);

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::ADD;
send_request(req);
wait_for_item_done();
//get_response(res);

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::POP_C;
send_request(req);
wait_for_item_done();
//get_response(res);

endtask
endclass

```

### Sequencer:

uvm\_sequence has a property called default\_sequence. Default sequence is a sequence which will be started automatically. Using set\_config\_string, user can override the default sequence to any user defined sequence, so that when a sequencer is started, automatically a user defined sequence will be started. If over rides are not done with user defined sequence, then a random transaction are generated. Using "start\_default\_sequence()" method, "default\_sequence" can also be started.

uvm sequencer has seq\_item\_export and res\_export tlm ports for connecting to uvm driver.

1) Define instruction\_sequencer by extending uvm\_sequencer.

```
class instruction_sequencer extends uvm_sequencer #(instruction);
```

2) Define the constructor.

Inside the constructor, place the macro `uvm\_update\_sequence\_lib\_and\_item().

This macro creates 3 predefined sequences. We will discuss about the predefined sequences in next section.

```
function new (string name, uvm_component parent);
  super.new(name, parent);
```

```

`uvm_update_sequence_lib_and_item(instruction)
endfunction

```

3) Place the uvm\_sequencer\_utils macro. This macro registers the sequencer for factory overrides.

```
`uvm_sequencer_utils(instruction_sequencer)
```

#### Sequencer Code:

```
class instruction_sequencer extends uvm_sequencer #(instruction);
```

```

function new (string name, uvm_component parent);
    super.new(name, parent);
    `uvm_update_sequence_lib_and_item(instruction)
endfunction

```

```
`uvm_sequencer_utils(instruction_sequencer)
endclass
```

#### Driver:

uvm\_driver is a class which is extended from uvm\_componenet. This driver is used in pull mode. Pull mode means, driver pulls the transaction from the sequencer when it requires. uvm driver has 2 TLM ports.

- 1) Seq\_item\_port: To get a item from sequencer, driver uses this port. Driver can also send response back using this port.
- 2) Rsp\_port : This can also be used to send response back to sequencer.

#### Seq\_item\_port methods:

Method	Type	Description
Task get_next_item(output REQreq_arg);	Blocking	Retrieves the next available item from a sequence.
task try_next_item( output REQreq_arg )	Non blocking	Retrieves the next available item from a sequence if one is available.
void item_done(RSPrsp_arg = null)	Non Blocking	Indicates that the request is completed to the sequencer.
task get( output REQreq_arg )	Blocking	Retrieves the next available item from a sequence.
task put( RSPrsp_arg )	Non Blocking	Sends a response back to the sequence that issued the request.
task peek( output REQreq_arg )	Blocking	Returns the current request item if one is in the sequencer fifo.

Lets implement a driver:

1) Define a driver which takes the instruction from the sequencer and does the processing. In this example we will just print the instruction type and wait for some delay.

```
class instruction_driver extends uvm_driver #(instruction);
```

2) Place the uvm\_component\_utils macro to define virtual methods like get\_type\_name and create.

```
`uvm_component_utils(instruction_driver)
```

3) Define Constructor method.

```
function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction
```

4) Define the run() method. Run() method is executed in the "run phase". In this methods, transactions are taken from the sequencer and drive them on to dut interface or to other components.

Driver class has a port "seq\_item\_port". Using the method seq\_item\_port.get\_next\_item(), get the transaction from the sequencer and process it. Once the processing is done, using the item\_done() method, indicate to the sequencer that the request is completed. In this example, after taking the transaction, we will print the transaction and wait for 10 units time.

```
task run ();
    while(1) begin
        seq_item_port.get_next_item(req);
        $display("%0d: Driving Instruction %s",$time,req.inst.name());
        #10;
        seq_item_port.item_done();
    end
endtask
endclass
```

#### Driver class code:

```
class instruction_driver extends uvm_driver #(instruction);

// Provide implementations of virtual methods such as get_type_name and create
`uvm_component_utils(instruction_driver)
// Constructor
function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction
task run ();
    forever begin
        seq_item_port.get_next_item(req);
        $display("%0d: Driving Instruction %s",$time,req.inst.name());
    end
endtask
```

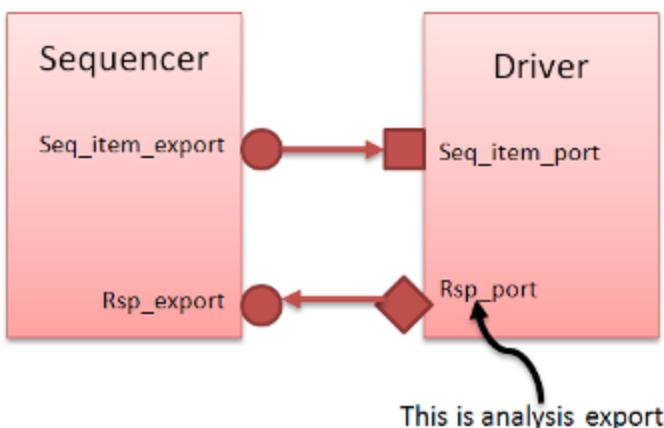
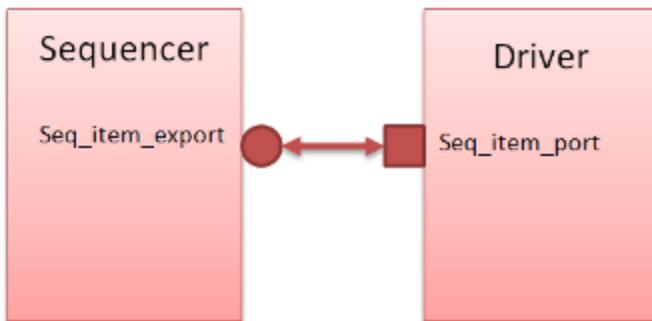
```

#10;
// rsp.set_id_info(req); These two steps are required only if
// seq_item_port.put(esp); response needs to be sent back to sequence
seq_item_port.item_done();
end
endtask
endclass

```

### Driver And Sequencer Connectivity:

Deriver and sequencer are connected using TLM. uvm\_driver has seq\_item\_port which is used to get the transaction from uvm sequencer. This port is connected to uvm\_sequencer seq\_item\_export Using "<driver>.seq\_item\_port.connect(<sequencer>.seq\_item\_export);" driver and sequencer can be connected. Simillarly "res\_port" of driver which is used to send response from driver to sequencer is connected to "res\_export" of the sequencer using ""<driver>.res\_port.connect(<sequencer>.res\_export);".



### Testcase:

This testcase is used only for the demo purpose of this tutorial session. Actually, the sequencer and the driver are instantiated and their ports are connected in a agent component and used. Lets

implement a testcase

- 1) Take instances of sequencer and driver and construct both components.

```
sequencer = new("sequencer", null);
sequencer.build();
driver = new("driver", null);
driver.build();
```

- 2)

Connect the seq\_item\_export to the drivers seq\_item\_port.

```
driver.seq_item_port.connect(sequencer.seq_item_export);
```

- 3) Using set\_cfg\_string() method, set the default sequence of the sequencer to "operation\_addition". Operation\_addition is the sequence which we defined previous.

```
set_config_string("sequencer", "default_sequence", "operation_addition");
```

- 4) Using the start\_default\_sequence() method of the sequencer, start the default sequence of the sequencer. In the previous step we configured the addition operation as default sequence. When you run the simulation, you will see the PUSH\_A,PUSH\_B ADD and POP\_C series of transaction.

```
sequencer.start_default_sequence();
```

#### Testcase Code:

```
module test;
instruction_sequencer sequencer;
instruction_driver driver;
initial begin
    set_config_string("sequencer", "default_sequence", "operation_addition");
    sequencer = new("sequencer", null);
    sequencer.build();
    driver = new("driver", null);
    driver.build();
    driver.seq_item_port.connect(sequencer.seq_item_export);
    sequencer.print();
    fork
        begin
            run_test();
            sequencer.start_default_sequence();
        end
        #2000 global_stop_request();
    join
end
endmodule
```

### Download the example:

[uvm\\_basic\\_sequence.tar](#)

[Browse the code in uvm\\_basic\\_sequence.tar](#)

### Command to simulate

VCS Users : make vcs

Questa Users: make questa

### Log file Output

UVM\_INFO @ 0 [RNTST] Running test ...

0: Driving Instruction PUSH\_A

10: Driving Instruction PUSH\_B

20: Driving Instruction ADD

30: Driving Instruction POP\_C

From the above log , we can see that transactions are generates as we defined in uvm sequence.

### Pre Defined Sequences:

Every sequencer in uvm has 3 pre defined sequences. They are

- ② 1)uvm\_random\_sequence
- ② 2)uvm\_exhaustive\_sequence.
- ② 3)uvm\_simple\_sequence

All the user defined sequences which are registered by user and the above three predefined sequences are stored in sequencer queue.

<b>id</b>	<b>Sequences[\$]</b>
0	uvm_random_sequence
1	uvm_exhaustive_sequence
2	uvm_simple_sequence
3	Userdefined sequence 1
4	Userdefined sequence 2
5	Userdefined sequence 3
.	...
.	...

© www.testbench.in

### uvm\_random\_sequence :

This sequence randomly selects and executes a sequence from the sequencer sequence library, excluding uvm\_random\_sequence itself, and uvm\_exhaustive\_sequence. From the above image, from sequence id 2 to till the last sequence, all the sequences are executed randomly. If the "count" variable of the sequencer is set to 0, then non of the sequence is executed. If the "count" variable of the sequencer is set to -1, then some random number of sequences from 0 to "max\_random\_count" are executed. By default "max\_random\_count" is set to 10. "Count" and

"max\_random\_count" can be changed using set\_config\_int().

The sequencer when automatically started executes the sequence which is point by default\_sequence. By default default\_sequence variable points to uvm\_random\_sequence.

#### uvm\_exhaustive\_sequence:

This sequence randomly selects and executes each sequence from the sequencers sequence library once in a randc style, excluding itself and uvm\_random\_sequence.

#### uvm\_simple\_sequence:

This sequence simply executes a single sequence item.

In the previous example from UVM\_SEQUENCE\_1 section.

The print() method of the sequencer in that example printed the following

Name	Type	Size	Value
sequencer	instruction_sequen+ -		sequencer@2
rsp_export	uvm_analysis_export -		rsp_export@4
seq_item_export	uvm_seq_item_pull_+ -		seq_item_export@28
default_sequence	string	18	operation_addition
count	integral	32	-1
max_random_count	integral	32	'd10
sequences	array	4	-
[0]	string	19	uvm_random_sequence
[1]	string	23	uvm_exhaustive_sequ+
[2]	string	19	uvm_simple_sequence
[3]	string	18	operation_addition
max_random_depth	integral	32	'd4
num_last_reqs	integral	32	'd1
num_last_rsps	integral	32	'd1

Some observations from the above log:

The count is set to -1. The default sequencer is set to operations\_addition. There are 3 predefined sequences and 1 user defined sequence.

Lets look at a example: In the attached example, in file sequence.sv file, there are 4 sequences, they are operation\_addition, operation\_subtraction, operation\_multiplication.

In the testcase.sv file, the "default\_sequence" is set to "uvm\_exhaustive\_sequence" using the set\_config\_string.

```
set_config_string("sequencer", "default_sequence", "uvm_exhaustive_sequence");
```

### Download the example

[uvm\\_sequence\\_1.tar](#)

[Browse the code in uvm\\_sequence\\_1.tar](#)

### Command to run the summation

VCS Users : make vcs

Questa Users: make questa

### Log File

0: Driving Instruction PUSH\_B

10: Driving Instruction PUSH\_A

20: Driving Instruction PUSH\_B

30: Driving Instruction SUB

40: Driving Instruction POP\_C

50: Driving Instruction PUSH\_A

60: Driving Instruction PUSH\_B

70: Driving Instruction MUL

80: Driving Instruction POP\_C

90: Driving Instruction PUSH\_A

100: Driving Instruction PUSH\_B

110: Driving Instruction ADD

120: Driving Instruction POP\_C

From the above log , we can see that all the 3 user defined sequences and predefined uvm\_simple\_sequence are executed.

### Sequence Action Macro:

In the previous sections, we have seen the implementation of body() method of sequence. The body() method implementation requires some steps. We have seen these steps as Creation of item, wait for grant, randomize the item, send the item.

All these steps have been automated using "sequence action macros". There are some more additional steps added in these macros. Following are the steps defined with the "sequence action macro".

	<b>Phase</b>	<b>Operation</b>
1	Create	Construct a transaction using create() method.
2	Synchronize	Call wait_for_grant() method
3	Pre_do	Call pre_do() method.
4	Randomize	Optionally Randomize transaction.
5	Mid_do	Call mid_do() method.
6	Post_Synchronize	Call send_request() and wait_for_item_done() methods.
7	Post_do	Optionally call post_do() and get_response() methods.

Pre\_do(), mid\_do() and post\_do() are callback methods which are in uvm sequence. If user is interested , he can use these methods. For example, in mid\_do() method, user can print the transaction or the randomized transaction can be fined tuned. These methods should not be called by user directly.

#### Syntax:

```
virtual task pre_do(bit is_item)
virtual function void mid_do(uvm_sequence_item this_item)
virtual function void post_do(uvm_sequence_item this_item)
```

Pre\_do() is a task , if the method consumes simulation cycles, the behavior may be unexpected.

#### Example Of Pre do,Mid do And Post do

Lets look at a example: We will define a sequence using `uvm\_do macro. This macro has all the above defined phases.

1)Define the body method using the `uvm\_do() macro. Before and after this macro, just call messages.

```
virtual task body();
  uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : Before uvm_do macro ",UVM_LOW);
  `uvm_do(req);
  uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : After uvm_do macro ",UVM_LOW);
endtask
```

2)Define pre\_do() method. Lets just print a message from this method.

```
virtual task pre_do(bit is_item);
  uvm_report_info(get_full_name(),
    "Seuqnce Action Macro Phase : PRE_DO ",UVM_LOW);
endtask
```

3)Define mid\_do() method. Lets just print a message from this method.

```
virtual function void mid_do(uvm_sequence_item this_item);
    uvm_report_info(get_full_name(),
        "Seuqnce Action Macro Phase : MID_DO ",UVM_LOW);
endfunction
```

4)Define post\_do() method. Lets just print a message from this method.

```
virtual function void post_do(uvm_sequence_item this_item);
    uvm_report_info(get_full_name(),
        "Seuqnce Action Macro Phase : POST_DO ",UVM_LOW);
endfunction
```

#### Complet sequence code:

```
class demo_uvm_do extends uvm_sequence #(instruction);
    instruction req;
    function new(string name="demo_uvm_do");
        super.new(name);
    endfunction
    `uvm_sequence_utils(demo_uvm_do, instruction_sequencer)
    virtual task pre_do(bit is_item);
        uvm_report_info(get_full_name(),
            "Seuqnce Action Macro Phase : PRE_DO ",UVM_LOW);
    endtask
    virtual function void mid_do(uvm_sequence_item this_item);
        uvm_report_info(get_full_name(),
            "Seuqnce Action Macro Phase : MID_DO ",UVM_LOW);
    endfunction
    virtual function void post_do(uvm_sequence_item this_item);
        uvm_report_info(get_full_name(),
            "Seuqnce Action Macro Phase : POST_DO ",UVM_LOW);
    endfunction
    virtual task body();
        uvm_report_info(get_full_name(),
            "Seuqnce Action Macro Phase : Before uvm_do macro ",UVM_LOW);
        `uvm_do(req);
        uvm_report_info(get_full_name(),
            "Seuqnce Action Macro Phase : After uvm_do macro ",UVM_LOW);
    endtask
endclass
```

## Download the example

[uvm\\_sequence\\_2.tar](#)

[Browse the code in uvm\\_sequence\\_2.tar](#)

## Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

## Log file report:

```
UVM_INFO@0:reporter[sequencer.demo_uvm_do]
  Sequence Action Macro Phase : Before uvm_do macro
UVM_INFO@0:reporter[sequencer.demo_uvm_do]
  Sequence Action Macro Phase : PRE_DO
UVM_INFO@0:reporter[sequencer.demo_uvm_do]
  Sequence Action Macro Phase : MID_DO
0: Driving Instruction MUL
UVM_INFO@10:reporter[sequencer.demo_uvm_do]
  Sequence Action Macro Phase : POST_DO
UVM_INFO@10:reporter[sequencer.demo_uvm_do]
  Sequence Action Macro Phase : After uvm_do macro
```

The above log file shows the messages from pre\_do,mid\_do and post\_do methods.

## List Of Sequence Action Macros:

These macros are used to start sequences and sequence items that were either registered with a <`uvm-sequence\_utils> macro or whose associated sequencer was already set using the <set\_sequencer> method.

### `uvm\_create(item/sequence)

This action creates the item or sequence using the factory. Only the create phase will be executed.

### `uvm\_do(item/sequence)

This macro takes as an argument a uvm\_sequence\_item variable or sequence . All the above defined 7 phases will be executed.

### `uvm\_do\_with(item/sequence, Constraint block)

This is the same as `uvm\_do except that the constraint block in the 2nd argument is applied to the item or sequence in a randomize with statement before execution.

### `uvm\_send(item/sequence)

Create phase and randomize phases are skipped, rest all the phases will be executed. Using `uvm\_create, create phase can be executed. Essentially, an `uvm\_do without the create or randomization.

### `uvm\_rand\_send(item/sequence)

Only create phase is skipped. rest of all the phases will be executed. User should use `uvm\_create to create the sequence or item.

### `uvm\_rand\_send with(item/sequence , Constraint block)

Only create phase is skipped. rest of all the phases will be executed. User should use `uvm\_create to create the sequence or item. Constraint block will be applied which randomization.

### `uvm\_do\_pri(item/sequence, priority )

This is the same as `uvm\_do except that the sequence item or sequence is executed with the priority specified in the argument.

### `uvm\_do\_pri with(item/sequence , constraint block , priority)

This is the same as `uvm\_do\_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

### `uvm\_send\_pri(item/sequence,priority)

This is the same as `uvm\_send except that the sequence item or sequence is executed with the priority specified in the argument.

### `uvm\_rand\_send\_pri(item/sequence,priority)

This is the same as `uvm\_rand\_send except that the sequence item or sequence is executed with the priority specified in the argument.

### `uvm\_rand\_send\_pri with(item/sequence,priority,constraint block)

This is the same as `uvm\_rand\_send\_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

Following macros are used on sequence or sequence items on a different sequencer.

### `uvm\_create\_on(item/sequence,sequencer)

This is the same as `uvm\_create except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

### `uvm\_do\_on(item/sequence,sequencer)

This is the same as `uvm\_do except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

### `uvm\_do\_on\_pri(item/sequence,sequencer, priority)

This is the same as `uvm\_do\_pri except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

### `uvm\_do\_on\_with(item/sequence,sequencer, constraint block)

This is the same as `uvm\_do\_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument. The user must supply brackets around the constraints.

### `uvm\_do\_on\_pri\_with(item/sequence,sequencer,priority,constraint block)

This is the same as `uvm\_do\_pri\_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

## Examples With Sequence Action Macros:

**virtual task** body();

(get\_full\_name(),

    "Executing Sequence Action Macro uvm\_do",UVM\_LOW);

```

`uvm_do(req)
endtask
virtual task body();
  uvm_report_info(get_full_name(),
    "Executing Sequence Action Macro uvm_do_with ",UVM_LOW);
  `uvm_do_with(req,{ inst == ADD; })
endtask
virtual task body();
  uvm_report_info(get_full_name(),
    "Executing Sequence Action Macro uvm_create and uvm_send",UVM_LOW);
  `uvm_create(req)
  req.inst = instruction::PUSH_B;
  `uvm_send(req)
endtask
virtual task body();
  uvm_report_info(get_full_name(),
    "Executing Sequence Action Macro uvm_create and uvm_rand_send",UVM_LOW);
  `uvm_create(req)
  `uvm_rand_send(req)
endtask

```

## Download the example

[uvm\\_sequence\\_3.tar](#)

[Browse the code in uvm\\_sequence\\_3.tar](#)

## Command to sun the simulation

VCS Users : make vcs

Questa Users: make questa

## Log file report

0: Driving Instruction PUSH\_B

UVM\_INFO@10:reporter[\*\*\*]Executing Sequence Action Macro uvm\_do\_with

10: Driving Instruction ADD

UVM\_INFO@20:reporter[\*\*\*]Executing Sequence Action Macro uvm\_create and uvm\_send

20: Driving Instruction PUSH\_B

UVM\_INFO@30:reporter[\*\*\*]Executing Sequence Action Macro uvm\_do

30: Driving Instruction DIV

UVM\_INFO@40:reporter[\*\*\*]Executing Sequence Action Macro uvm\_create and  
uvm\_rand\_send

40: Driving Instruction MUL

## **UVM SEQUENCE 3**

### **Body Callbacks:**

uvm sequences has two callback methods pre\_body() and post\_body(), which are executed before and after the sequence body() method execution. These callbacks are called only when start\_sequence() of sequencer or start() method of the sequence is called. User should not call these methods.

**virtual task** pre\_body()  
**virtual task** post\_body()

### **Example**

In this example, I just printed messages from pre\_body() and post\_body() methods. These methods can be used for initialization, synchronization with some events or cleanup.

```
class demo_pre_body_post_body extends uvm_sequence #(instruction);
instruction req;
function new(string name="demo_pre_body_post_body");
super.new(name);
endfunction
`uvm_sequence_utils(demo_pre_body_post_body, instruction_sequencer)
virtual task pre_body();
    uvm_report_info(get_full_name()," pre_body() callback ",UVM_LOW);
endtask

virtual task post_body();
    uvm_report_info(get_full_name()," post_body() callback ",UVM_LOW);
endtask

virtual task body();
    uvm_report_info(get_full_name(),
        "body() method: Before uvm_do macro ",UVM_LOW);
    `uvm_do(req);
    uvm_report_info(get_full_name(),
        "body() method: After uvm_do macro ",UVM_LOW);
endtask

endclass
```

### **Download the example**

[uvm\\_sequence\\_4.tar](#)

[Browse the code in uvm\\_sequence\\_4.tar](#)

## Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

### Log file report

```
UVM_INFO @ 0 [RNTST] Running test ...
UVM_INFO @ 0: reporter [***] pre_body() callback
UVM_INFO @ 0: reporter [***] body() method: Before uvm_do macro
0: Driving Instruction SUB
UVM_INFO @ 10: reporter [***] body() method: After uvm_do macro
UVM_INFO @ 10: reporter [***] post_body() callback
```

### Hierarchical Sequences

One main advantage of sequences is smaller sequences can be used to create sequences to generate stimulus required for todays complex protocol.

To create a sequence using another sequence, following steps has to be done

- 1)Extend the uvm\_sequence class and define a new class.
- 2)Declare instances of child sequences which will be used to create new sequence.
- 3)Start the child sequence using <instance>.start() method in body() method.

### Sequential Sequences

To executes child sequences sequentially, child sequence start() method should be called sequentially in body method.

In the below example you can see all the 3 steps mentioned above.

In this example, I have defined 2 child sequences. These child sequences can be used as normal sequences.

#### Sequence 1 code:

This sequence generates 4 PUSH\_A instructions.

```
virtual task body();
    repeat(4) begin
        `uvm_do_with(req, { inst == PUSH_A; });
    end
endtask
```

#### Sequence 2 code:

This sequence generates 4 PUSH\_B instructions.

```
virtual task body();
    repeat(4) begin
        `uvm_do_with(req, { inst == PUSH_B; });
    end
endtask
```

### Sequential Sequence code:

This sequence first calls sequence 1 and then calls sequence 2.

```
class sequential_sequence extends uvm_sequence #(instruction);
    seq_a s_a;
    seq_b s_b;
    function new(string name="sequential_sequence");
        super.new(name);
    endfunction
    `uvm_sequence_utils(sequential_sequence, instruction_sequencer)
    virtual task body();
        `uvm_do(s_a);
        `uvm_do(s_b);
    endtask
endclass
```

From the testcase, "sequential\_sequence" is selected as "default\_sequence".

### Download the example

[uvm\\_sequence\\_5.tar](#)

[Browse the code in uvm\\_sequence\\_5.tar](#)

### Command to sun the simulation

VCS Users : make vcs

Questa Users: make questa

### Log file report

```
0: Driving Instruction PUSH_A
10: Driving Instruction PUSH_A
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_A
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction PUSH_B
```

If you observe the above log, you can see sequence seq\_a is executed first and then sequene seq\_b is executed.

### Parallel sequences

To executes child sequences Parallel, child sequence start() method should be called parallel using fork/join in body method.

### Parallel Sequence code:

```
class parallel_sequence extends uvm_sequence #(instruction);
    seq_a s_a;
    seq_b s_b;
```

```

function new(string name="parallel_sequence");
    super.new(name);
endfunction

`uvm_sequence_utils(parallel_sequence, instruction_sequencer)
virtual task body();
    fork
        `uvm_do(s_a)
        `uvm_do(s_b)
    join
endtask
endclass

```

[Download the example](#)

[uvm\\_sequence\\_6.tar](#)

[Browse the code in uvm\\_sequence\\_6.tar](#)

[Command to sun the simulation](#)

VCS Users : make vcs

Questa Users: make questa

[Log file report](#)

UVM\_INFO @ 0 [RNTST] Running test ...

```

0: Driving Instruction PUSH_A
10: Driving Instruction PUSH_B
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_A
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_A
70: Driving Instruction PUSH_B

```

## **UVM SEQUENCE 4**

### **Sequencer Arbitration:**

When sequences are executed parallel, sequencer will arbitrate among the parallel sequence.

When all the parallel sequences are waiting for a grant from sequencer using wait\_for\_grant() method, then the sequencer, using the arbitration mechanism, sequencer grants to one of the sequencer.

There are 6 different arbitration algorithms, they are

Macro	Algorithm
SEQ_ARB_FIFO	Requests are granted in FIFO style. Priorities will not be considered.
SEQ_ARB_RANDOM	Requests are granted randomly. Priorities will not be considered.
SEQ_ARB_WEIGHTED	Based on the priority value, randomly grants the requests.
SEQ_ARB_STRICT_FIFO	Sequences with high priority value will be granted in FIFO style.
SEQ_ARB_STRICT_RANDOM	Sequences with high priority value will be granted randomly.
SEQ_ARB_USER	Grants are done based on user defined algorithm.

© www.testbench.in

To set the arbitration, use the `set_arbitration()` method of the sequencer. By default , the arbitration algorithms is set to SEQ\_ARB\_FIFO.

**function void** set\_arbitration(**SEQ\_ARB\_TYPE** val)

Lets look at a example.

In this example, I have 3 child sequences seq\_mul seq\_add and seq\_sub each of them generates 3 transactions.

Sequence code 1:

```
virtual task body();
    repeat(3) begin
        `uvm_do_with(req, { inst == MUL; });
    end
endtask
```

Sequence code 2:

```
virtual task body();
    repeat(3) begin
        `uvm_do_with(req, { inst == ADD; });
    end
endtask
```

Sequence code 3:

```
virtual task body();
    repeat(3) begin
        `uvm_do_with(req, { inst == SUB; });
    end
endtask
```

### Parallel sequence code:

In the body method, before starting child sequences, set the arbitration using set\_arbitration(). In this code, im setting it to SEQ\_ARB\_RANDOM.

```
class parallel_sequence extends uvm_sequence #(instruction);
    seq_add add;
    seq_sub sub;
    seq_mul mul;
    function new(string name="parallel_sequence");
        super.new(name);
    endfunction
    `uvm_sequence_utils(parallel_sequence, instruction_sequencer)

    virtual task body();
        m_sequencer.set_arbitration(SEQ_ARB_RANDOM);
        fork
            `uvm_do(add)
            `uvm_do(sub)
            `uvm_do(mul)
        join
    endtask
endclass
```

### Download the example

[uvm\\_sequence\\_7.tar](#)

[Browse the code in uvm\\_sequence\\_7.tar](#)

### Command to sun the simulation

VCS Users : make vcs

Questa Users: make questa

### Log file report for when SEQ\_ARB\_RANDOM is set.

```
0: Driving Instruction MUL
10: Driving Instruction SUB
20: Driving Instruction MUL
30: Driving Instruction SUB
40: Driving Instruction MUL
50: Driving Instruction ADD
60: Driving Instruction ADD
70: Driving Instruction SUB
80: Driving Instruction ADD
```

### Log file report for when SEQ\_ARB\_FIFO is set.

```
0: Driving Instruction ADD
10: Driving Instruction SUB
20: Driving Instruction MUL
```

```
30: Driving Instruction ADD  
40: Driving Instruction SUB  
50: Driving Instruction MUL  
60: Driving Instruction ADD  
70: Driving Instruction SUB  
80: Driving Instruction MUL
```

If you observe the first log report, all the transaction of the sequences are generated in random order. In the second log file, the transactions are given equal priority and are in fifo order.

### **Setting The Sequence Priority:**

There are two ways to set the priority of a sequence. One is using the start method of the sequence and other using the set\_priority() method of the sequence. By default, the priority of a sequence is 100. Higher numbers indicate higher priority.

```
virtual task start (uvm_sequencer_base sequencer,  
    uvm_sequence_base parent_sequence = null,  
    integer this_priority = 100,  
    bit call_pre_post = 1)
```

```
function void set_priority (int value)
```

Lets look a example with SEQ\_ARB\_WEIGHTED.

For sequence seq\_mul set the weight to 200.

For sequence seq\_add set the weight to 300.

For sequence seq\_sub set the weight to 400.

In the below example, start() method is used to override the default priority value.

### **Code :**

```
class parallel_sequence extends uvm_sequence #(instruction);  
    seq_add add;  
    seq_sub sub;  
    seq_mul mul;  
    function new(string name="parallel_sequence");  
        super.new(name);  
    endfunction  
    `uvm_sequence_utils(parallel_sequence, instruction_sequencer)  
    virtual task body();  
        m_sequencer.set_arbitration(SEQ_ARB_WEIGHTED);  
        add = new("add");  
        sub = new("sub");  
        mul = new("mul");  
        fork  
            sub.start(m_sequencer, this, 400);  
            add.start(m_sequencer, this, 300);
```

```
mul.start(m_sequencer,this,200);
join
endtask
endclass
```

### Download the example

[uvm\\_sequence\\_8.tar](#)

[Browse the code in uvm\\_sequence\\_8.tar](#)

### Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

### Log file report

```
0: Driving Instruction MUL
10: Driving Instruction ADD
20: Driving Instruction SUB
30: Driving Instruction SUB
40: Driving Instruction ADD
50: Driving Instruction ADD
60: Driving Instruction ADD
70: Driving Instruction MUL
80: Driving Instruction SUB
```

## **UVM SEQUENCE 5**

### Sequencer Registration Macros

Sequencer Registration Macros does the following

- 1) Implements get\_type\_name method.
- 2) Implements create() method.
- 3) Registers with the factory.
- 4) Implements the static get\_type() method.
- 5) Implements the virtual get\_object\_type() method.
- 6) Registers the sequence type with the sequencer type.
- 7) Defines p\_sequencer variable. p\_sequencer is a handle to its sequencer.
- 8) Implements m\_set\_p\_sequencer() method.

If there are no local variables, then use following macro

``uvm_sequence_utils(TYPE_NAME,SQR_TYPE_NAME)`

If there are local variables in sequence, then use macro

``uvm_sequence_utils_begin(TYPE_NAME,SQR_TYPE_NAME)`

``uvm_field_*` macro invocations here

``uvm_sequence_utils_end`

Macros `uvm\_field\_\* are used for define utility methods.

These `uvm\_field\_\* macros are discussed in

### UVM TRANSACTION

Example to demonstrate the usage of the above macros:

```
class seq_mul extends uvm_sequence #(instruction);
  rand integer num_inst ;
  instruction req;
  constraint num_c { num_inst inside { 3,5,7 } ; };
  `uvm_sequence_utils_begin(seq_mul,instruction_sequencer)
  `uvm_field_int(num_inst, UVM_ALL_ON)
  `uvm_sequence_utils_end

  function new(string name="seq_mul");
    super.new(name);
  endfunction
  virtual task body();
    uvm_report_info(get_full_name(),
      $psprintf("Num of transactions %d",num_inst),UVM_LOW);
    repeat(num_inst) begin
      `uvm_do_with(req, { inst == MUL; });
    end
  endtask
endclass
```

### Download the example

[uvm\\_sequence\\_9.tar](#)

[Browse the code in uvm\\_sequence\\_9.tar](#)

### Command to sun the simulation

VCS Users : make vcs

Questa Users: make questa

### Log

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
UVM_INFO @ 0: reporter [sequencer.seq_mul] Num of transactions      5
0: Driving Instruction MUL
10: Driving Instruction MUL
20: Driving Instruction MUL
30: Driving Instruction MUL
40: Driving Instruction MUL
```

## Setting Sequence Members:

set\_config\_\* can be used only for the components not for the sequences. By using configuration you can change the variables inside components only not in sequences. But there is a workaround to this problem.

Sequence has handle name called p\_sequencer which is pointing the Sequencer on which it is running.

Sequencer is a component , so get\_config\_\* methods are implemented for it. So from the sequence, using the sequencer get\_config\_\* methods, sequence members can be updated if the variable is configured.

When using set\_config\_\* , path to the variable should be sequencer name, as we are using the sequencer get\_config\_\* method.

Following method demonstrates how this can be done:

### Sequence:

- 1) num\_inst is a integer variables which can be updated.
- 2) In the body method, call the get\_config\_int() method to get the integer value if num\_inst is configured from testcase.

```
class seq_mul extends uvm_sequence #(instruction);
  integer num_inst = 4;
  instruction req;
  `uvm_sequence_utils_begin(seq_mul,instruction_sequencer)
    `uvm_field_int(num_inst, UVM_ALL_ON)
  `uvm_sequence_utils_end
  function new(string name="seq_mul");
    super.new(name);
  endfunction
  virtual task body();
    void'(p_sequencer.get_config_int("num_inst",num_inst));
    uvm_report_info(get_full_name(),
      $psprintf("Num of transactions %d",num_inst),UVM_LOW);
    repeat(num_inst) begin
      `uvm_do_with(req, { inst == MUL; });
    end
  endtask
endclass
```

### Testcase:

From the testcase, using the set\_config\_int() method, configure the num\_inst to 3.  
The instance path argument should be the sequencer path name.

```
module test;
```

```

instruction_sequencer sequencer;
instruction_driver driver;
initial begin
    set_config_string("sequencer", "default_sequence", "seq_mul");
    set_config_int("sequencer", "num_inst",3);
    sequencer = new("sequencer", null);
    sequencer.build();
    driver = new("driver", null);
    driver.build();

    driver.seq_item_port.connect(sequencer.seq_item_export);
    sequencer.print();
fork
    begin
        run_test();
        sequencer.start_default_sequence();
    end
    #3000 global_stop_request();
    join
end
endmodule

```

[Download the example](#)

[uvm\\_sequence\\_10.tar](#)

[Browse the code in uvm\\_sequence\\_10.tar](#)

[Command to sun the simulation](#)

VCS Users : make vcs

Questa Users: make questa

[Log](#)

UVM\_INFO @ 0: reporter [RNTST] Running test ...

UVM\_INFO @ 0: reporter [sequencer.seq\_mul] Num of transactions

3

0: Driving Instruction MUL

10: Driving Instruction MUL

20: Driving Instruction MUL

From the above log we can see that seq\_mul.num\_inst value is 3.

## **UVM SEQUENCE 6**

### **Exclusive Access**

A sequence may need exclusive access to the driver which sequencer is arbitrating among multiple sequence. Some operations require that a series of transaction needs to be driven without any other transaction in between them. Then a exclusive access to the driver will allow to a sequence to complete its operation with out any other sequence operations in between them. There are 2 mechanisms to get exclusive access:

- ⌚ Lock-unlcok
- ⌚ Grab-ungrab

### **Lock-Unlock**

```
task lock(uvm_sequencer_base sequencer = Null)  
function void unlock(uvm_sequencer_base sequencer = Null)
```

Using lock() method , a sequence can requests for exclusive access. A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence. A lock() is blocking task and when access is granted, it will unblock.

Using unlock(), removes any locks or grabs obtained by this sequence on the specified sequencer.

If sequencer is null, the lock/unlock will be applied on the current default sequencer.

Lets see an example,

In this example there are 3 sequences with each sequence generating 4 transactions. All these 3 sequences will be called in parallel in another sequence.

#### **Sequence 1 code:**

```
virtual task body();  
    repeat(4) begin  
        `uvm_do_with(req, { inst == PUSH_A; });  
    end  
endtask
```

#### **Sequence 2 code:**

```
virtual task body();  
    repeat(4) begin  
        `uvm_do_with(req, { inst == POP_C; });  
    end  
endtask
```

**Sequence 3 code:** In this sequence , call the lock() method to get the exclusive access to driver. After completing all the transaction driving, then call the unclock() method.

```
virtual task body();  
    lock();
```

```

repeat(4) begin
  `uvm_do_with(req, { inst == PUSH_B; });
end
unlock();
endtask

```

#### Parallel sequence code:

```
virtual task body();
```

```

fork
  `uvm_do(s_a)
  `uvm_do(s_b)
  `uvm_do(s_c)

```

```
join
```

```
endtask
```

#### Download the example

[uvm\\_sequence\\_11.tar](#)

[Browse the code in uvm\\_sequence\\_11.tar](#)

#### Command to sun the simulation

VCS Users : make vcs

Questa Users: make questa

#### Log file:

```

0: Driving Instruction PUSH_A
10: Driving Instruction POP_C
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction POP_C
80: Driving Instruction PUSH_A
90: Driving Instruction POP_C
100: Driving Instruction PUSH_A
110: Driving Instruction POP_C

```

From the above log file, we can observe that , when seq\_b sequence got the access, then transactions from seq\_a and seq\_c are not generated.

Lock() will be arbitrated before giving the access. To get the exclusive access without arbitration, grab() method should be used.

#### Grab-Ungrab

```
task grab(uvm_sequencer_base sequencer = null)
```

```
function void ungrab(uvm_sequencer_base sequencer = null)
```

grab() method requests a lock on the specified sequencer. A grab() request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab() is granted when no other grabs or locks are blocking this sequence.

A grab() is blocking task and when access is granted, it will unblock.

Ungrab() method removes any locks or grabs obtained by this sequence on the specified sequencer.

If no argument is supplied, then current default sequencer is chosen.

Example:

```
virtual task body();
#25;
grab();
repeat(4) begin
`uvm_do_with(req, { inst == PUSH_B; });
end
ungrab();
endtask
```

[Download the example](#)

[uvm\\_sequence\\_12.tar](#)

[Browse the code in uvm\\_sequence\\_12.tar](#)

[Command to run the simulation](#)

VCS Users : make vcs

Questa Users: make questa

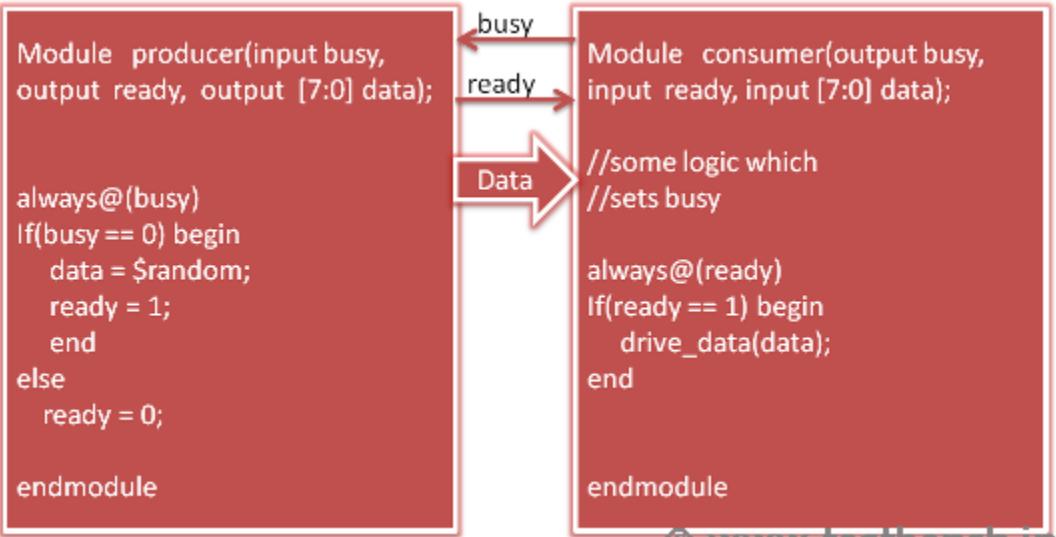
```
0: Driving Instruction PUSH_A
10: Driving Instruction POP_C
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction POP_C
80: Driving Instruction PUSH_A
90: Driving Instruction POP_C
100: Driving Instruction PUSH_A
110: Driving Instruction POP_C
```

## **UVM TLM 1**

Before going into the TLM interface concepts, lets see why we need TLM interface

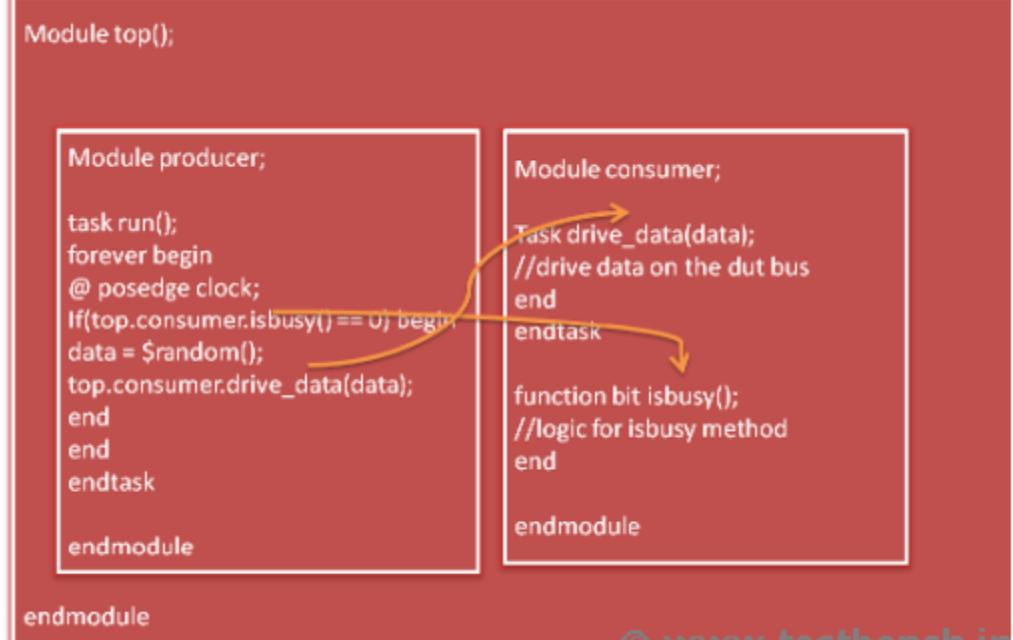
### **Port Based Data Transfer:**

Following is a simple verification environment.



Components generator and driver are implemented as modules. These modules are connected using module ports or SV interfaces. The advantage of this methodology is, the two above mentioned components are independent. Instead of consumer module, any other component which can understand producer interface can be connected, which gives a great reusability. The disadvantage of this methodology is , data transfer is done at lower lever abstraction.

### Task Based Data Transfer:



In the above environment, methods are used to transfer the data between components. So, this gives a better control and data transfer is done at high level. The disadvantage is, components are

using hierachal paths which do not allow the reusability.

### TLM interfaces:

UVM has TLM interfaces which provide the advantages which we saw in the above two data transfer styles.

Data is transferred at high level. Transactions which are developed by extending the uvm\_sequence\_item can be transferred between components using method calls. These methods are not hierachal fixed, so that components can be reused.

The advantages of TLM interfaces are

- ⌚ 1) Higher level abstraction
- ⌚ 2) Reusable. Plug and play connections.
- ⌚ 3) Maintainability
- ⌚ 4) Less code.
- ⌚ 5) Easy to implement.
- ⌚ 6) Faster simulation.
- ⌚ 7) Connect to Systemc.
- ⌚ 8) Can be used for reference model development.

### Operation Supported By Tlm Interface:

#### **⌚ Putting:**

Producer transfers a value to Consumer.

#### **⌚ Getting:**

Consumer requires a data value from producer.

#### **⌚ Peeking:**

Copies data from a producer without consuming the data.

#### **⌚ Broadcasting:**

Transaction is broadcasted to none or one or multiple consumers.

### Methods

#### BLOCKING:

```
virtual task put(input T1 t)  
virtual task get(output T2 t)  
virtual task peek(output T2 t)
```

#### NON-BLOCKIN:

```
virtual function bit try_put(input T1 t)  
virtual function bit can_put()  
virtual function bit try_get(output T2 t)  
virtual function bit can_get()  
virtual function bit try_peek(output T2 t)  
virtual function bit can_peek()
```

#### BLOCKING TRANSPORT:

```
virtual task transport(input T1 req,output T2 rsp)
```

## NON-BLOCKING TRANSPORT:

**virtual function bit nb\_transport(**

## ANALYSIS:

**virtual function void write(**

## Tlm Terminology :

### **⌚ Producer:**

A component which generates a transaction.

### **⌚ Consumer:**

A component which consumes the transaction.

### **⌚ Initiator:**

A component which initiates process.

### **⌚ Target:**

A component which responded to initiator.

## Tlm Interface Compilation Models:

### **⌚ Blocking:**

A blocking interface conveys transactions in blocking fashion; its methods do not return until the transaction has been successfully sent or retrieved. Its methods are defined as tasks.

### **⌚ Non-blocking:**

A non-blocking interface attempts to convey a transaction without consuming simulation time. Its methods are declared as functions. Because delivery may fail (e.g. the target component is busy and can not accept the request), the methods may return with failed status.

### **⌚ Combined:**

A combination interface contains both the blocking and non-blocking variants.

## Interfaces:

The UVM provides ports, exports and implementation and analysis ports for connecting your components via the TLM interfaces. Port, Export, implementation terminology applies to control flow not to data flow.

### **⌚ Port:**

Interface that requires an implementation is port.

### **⌚ Import:**

Interface that provides an implementation is import or implementation port.

### **⌚ Export:**

Interface used to route transaction interfaces to other layers of the hierarchy.

### **⌚ Analysis:**

Interface used to distribute transactions to passive components.

## Direction:

### ☞ Unidirectional:

Data transfer is done in a single direction and flow of control is in either or both direction.

### ☞ Bidirectional:

Data transfer is done in both directions and flow of control is in either or both directions.

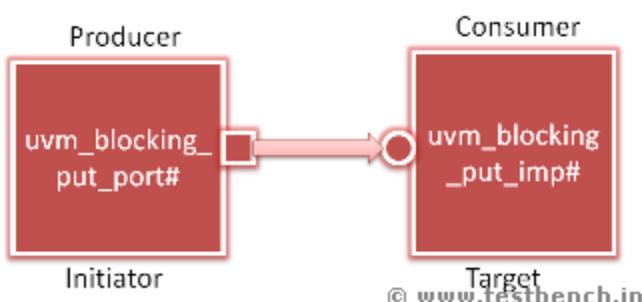
Examples:

A read operation is a bidirectional.

A write operation is unidirectional.

Lets look at a example:

In this example, we will use \*\_put\_\* interface.



There are 2 components, producer and consumer.

Producer generates the transaction and Consumers consumes it.

In this example, producer calls the put() method to send transaction to consumer i.e producer is initiator and consumer is target.

When the put() method in the producer is called, it actually executes the put() method which is defined in consumer component.

## Transaction

We will use the below transaction in this example.

```

class instruction extends uvm_sequence_item;
  typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
  rand inst_t inst;

  `uvm_object_utils_begin(instruction)
    `uvm_field_enum(inst_t,inst, UVM_ALL_ON)
  `uvm_object_utils_end

  function new (string name = "instruction");
    super.new(name);
  endfunction
endclass

```

### Producer:

1) Define producer component by extending uvm\_component.

```
class producer extends uvm_component;  
endclass : producer
```

2) Declare uvm\_blocking\_put\_port port.

```
uvm_blocking_put_port#(instruction) put_port;
```

3) In the constructor, construct the port.

```
function new(string name, uvm_component p = null);  
super.new(name,p);  
put_port = new("put_port", this);  
endfunction
```

4) Define the run() method. In this method, randomize the transaction.

Then call the put() of the put\_port and pass the randomized transaction.

```
task run;  
for(int i = 0; i < 10; i++)  
begin  
instruction ints;  
#10;  
ints = new();  
if(ints.randomize()) begin  
`uvm_info("producer", $sformatf("sending %s",ints.inst.name()), UVM_MEDIUM)  
put_port.put(ints);  
end  
end  
endtask
```

### Producer source code

```
class producer extends uvm_component;  
uvm_blocking_put_port#(instruction) put_port;  
function new(string name, uvm_component p = null);  
super.new(name,p);  
put_port = new("put_port", this);  
endfunction  
task run;  
for(int i = 0; i < 10; i++)  
begin  
instruction ints;  
#10;  
ints = new();  
if(ints.randomize()) begin  
`uvm_info("producer", $sformatf("sending %s",ints.inst.name()), UVM_MEDIUM)
```

```

    put_port.put(ints);
end
end
endtask
endclass : producer

```

### Consumer:

1) Define a consumer component by extending uvm\_component.

```

class consumer extends uvm_component;
endclass : consumer

```

2) Declare uvm\_blocking\_put\_imp import. The parameters to this port are transaction and the consumer component itself.

```
uvm_blocking_put_imp#(instruction,consumer) put_port;
```

3) In the construct , construct the port.

```

function new(string name, uvm_component p = null);
super.new(name,p);
put_port = new("put_port", this);
endfunction

```

4) Define put() method. When the producer calls "put\_port.put(ints);", then this method will be called. Arguments to this method is transaction type "instruction".

In this method, we will just print the transaction.

```

task put(instruction t);
`uvm_info("consumer", $sformatf("receiving %s",t.inst.name()), UVM_MEDIUM)
endtask

```

### Consumer source code

```

class consumer extends uvm_component;
uvm_blocking_put_imp#(instruction,consumer) put_port;
function new(string name, uvm_component p = null);
super.new(name,p);
put_port = new("put_port", this);
endfunction
task put(instruction t);
`uvm_info("consumer", $sformatf("receiving %s",t.inst.name()), UVM_MEDIUM)
//push the transaction into queue or array
//or drive the transaction to next level
//or drive to interface
endtask
endclass : consumer

```

### Connecting producer and consumer

In the env class, take the instance of producer and consumer components.

In the connect method, connect the producer put\_port to consumer put\_port using

```
p.put_port.connect(c.put_port);
```

#### Env Source code

```
class env extends uvm_env;
producer p;
consumer c;
function new(string name = "env");
super.new(name);
p = new("producer", this);
c = new("consumer", this);
endfunction
function void connect();
p.put_port.connect(c.put_port);
endfunction
task run();
#1000;
global_stop_request();
endtask
endclass
```

#### Testcase

```
module test;
env e;
initial begin
e = new();
run_test();
end
endmodule
```

#### Download the example

[uvm\\_tlm\\_1.tar](#)

[Browse the code in uvm\\_tlm\\_1.tar](#)

#### Command to sun the simulation

VCS Users : make vcs

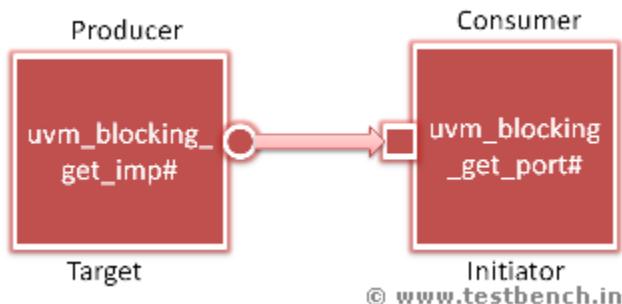
Questa Users: make questa

#### Log

```
UVM_INFO producer.sv(26) @ 10:
env.producer [producer] sending PUSH_A
UVM_INFO consumer.sv(20) @ 10:
env.consumer [consumer] receiving PUSH_A
UVM_INFO producer.sv(26) @ 20:
env.producer [producer] sending PUSH_B
UVM_INFO consumer.sv(20) @ 20:
```

env.consumer [consumer] receiving PUSH\_B

One more example using \*\_get\_\* interface as per the below topology.



### Download the example

[uvm\\_tlm\\_2.tar](#)

[Browse the code in uvm\\_tlm\\_2.tar](#)

### Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

### All Interfaces In Uvm:

```
uvm_blocking_put_port #(T)
uvm_nonblocking_put_port #(T)
uvm_put_port #(T)
uvm_blocking_get_port #(T)
uvm_nonblocking_get_port #(T)
uvm_get_port #(T)
uvm_blocking_peek_port #(T)
uvm_nonblocking_peek_port #(T)
uvm_peek_port #(T)
uvm_blocking_get_peek_port #(T)
uvm_nonblocking_get_peek_port #(T)
uvm_get_peek_port #(T)
uvm_analysis_port #(T)
uvm_transport_port #(REQ,RSP)
uvm_blocking_transport_port #(REQ,RSP)
uvm_nonblocking_transport_port #(REQ,RSP)
uvm_master_port #(REQ,RSP)
uvm_blocking_master_port #(REQ,RSP)
uvm_nonblocking_master_port #(REQ,RSP)
```

```
uvm_slave_port #(REQ,RSP)
uvm_blocking_slave_port #(REQ,RSP)
uvm_nonblocking_slave_port #(REQ,RSP)
uvm_put_export #(T)
uvm_blocking_put_export #(T)
uvm_nonblocking_put_export #(T)
uvm_get_export #(T)
uvm_blocking_get_export #(T)
uvm_nonblocking_get_export #(T)
uvm_peek_export #(T)
uvm_blocking_peek_export #(T)
uvm_nonblocking_peek_export #(T)
uvm_get_peek_export #(T)
uvm_blocking_get_peek_export #(T)
uvm_nonblocking_get_peek_export #(T)
uvm_analysis_export #(T)
uvm_transport_export #(REQ,RSP)
uvm_nonblocking_transport_export #(REQ,RSP)
uvm_master_export #(REQ,RSP)
uvm_blocking_master_export #(REQ,RSP)
uvm_nonblocking_master_export #(REQ,RSP)
uvm_slave_export #(REQ,RSP)
uvm_blocking_slave_export #(REQ,RSP)
uvm_nonblocking_slave_export #(REQ,RSP)
uvm_put_imp #(T,IMP)
uvm_blocking_put_imp #(T,IMP)
uvm_nonblocking_put_imp #(T,IMP)
uvm_get_imp #(T,IMP)
uvm_blocking_get_imp #(T,IMP)
uvm_nonblocking_get_imp #(T,IMP)
uvm_peek_imp #(T,IMP)
uvm_blocking_peek_imp #(T,IMP)
uvm_nonblocking_peek_imp #(T,IMP)
uvm_get_peek_imp #(T,IMP)
uvm_blocking_get_peek_imp #(T,IMP)
uvm_nonblocking_get_peek_imp #(T,IMP)
uvm_analysis_imp #(T,IMP)
uvm_transport_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_blocking_transport_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_nonblocking_transport_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_master_imp #(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_blocking_master_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_nonblocking_master_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_slave_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_blocking_slave_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_nonblocking_slave_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
```

## **UVM TLM 2**

### **Analysis**

The analysis port is used to perform non-blocking broadcasts of transactions. It is by components like monitors/drivers to publish transactions to its subscribers, which are typically scoreboards and response/coverage collectors. For each port, more than one component can be connected. Even if a component is not connected to the port, simulation can continue, unlike put/get ports where simulation is not continued.

The uvm\_analysis\_port consists of a single function, write(). Subscriber component should provide an implementation of write() method. UVM provides the uvm\_subscriber base component to simplify this operation, so a typical analysis component would extend uvm\_subscriber and its export is analysis\_export.

Lets write a example.

In the example, we will define a monitor component and a subscriber.

#### **Monitor source code:**

In monitor, call the function write() pass the transaction.

```
class monitor extends uvm_monitor;
    uvm_analysis_port #(instruction) anls_port;
    function new(string name, uvm_component p = null);
        super.new(name,p);
        anls_port = new("anls_port", this);
    endfunction
    task run;
        instruction inst;
        inst = new();
        #10ns;
        inst.inst = instruction::MUL;
        anls_port.write(inst);
        #10ns;
        inst.inst = instruction::ADD;
        anls_port.write(inst);
        #10ns;
        inst.inst = instruction::SUB;
        anls_port.write(inst);
    endtask
endclass
```

#### **Subscriber source code:**

In Subscriber, define the write() method.

```
class subscriber extends uvm_subscriber#(instruction);
    function new(string name, uvm_component p = null);
```

```

super.new(name,p);
endfunction
function void write(instruction t);
`uvm_info(get_full_name(),
 $sformatf("receiving %s",t.inst.name()), UVM_MEDIUM)
endfunction
endclass : subscriber
Env source code:
class env extends uvm_env;
monitor mon;
subscriber sb,cov;

function new(string name = "env");
super.new(name);
mon = new("mon", this);
sb = new("sb", this);
cov = new("cov", this);
endfunction

function void connect();
mon.anls_port.connect(sb.analysis_export);
mon.anls_port.connect(cov.analysis_export);
endfunction
task run();
#1000;
global_stop_request();
endtask
endclass
module test;
env e;
initial begin
e = new();
run_test();
end
endmodule

```

Download the example

[uvm\\_tlm\\_3.tar](#)

[Browse the code in uvm\\_tlm\\_3.tar](#)

Command to sun the simulation

VCS Users : make vcs

Questa Users: make questa

From the below log, you see that transaction is sent to both the components cov and sb.

### Log

```
UVM_INFO subscriber.sv(18) @ 0: env.cov [env.cov] receiving MUL
UVM_INFO subscriber.sv(18) @ 0: env.sb [env.sb] receiving MUL
UVM_INFO subscriber.sv(18) @ 0: env.cov [env.cov] receiving ADD
UVM_INFO subscriber.sv(18) @ 0: env.sb [env.sb] receiving ADD
UVM_INFO subscriber.sv(18) @ 0: env.cov [env.cov] receiving SUB
UVM_INFO subscriber.sv(18) @ 0: env.sb [env.sb] receiving SUB
```

### Tlm Fifo

Tlm\_fifo provides storage of transactions between two independently running processes just like mailbox. Transactions are put into the FIFO via the put\_export and fetched from the get\_export.

### Methods

Following are the methods defined for tlm fifo.

```
function new(string name,
  uvm_component parent = null,
  int size = 1)
```

The size indicates the maximum size of the FIFO; a value of zero indicates no upper bound.

```
virtual function int size()
```

Returns the capacity of the FIFO. 0 indicates the FIFO capacity has no limit.

```
virtual function int used()
```

Returns the number of entries put into the FIFO.

```
virtual function bit is_empty()
```

Returns 1 when there are no entries in the FIFO, 0 otherwise.

```
virtual function bit is_full()
```

Returns 1 when the number of entries in the FIFO is equal to its size, 0 otherwise.

```
virtual function void flush()
```

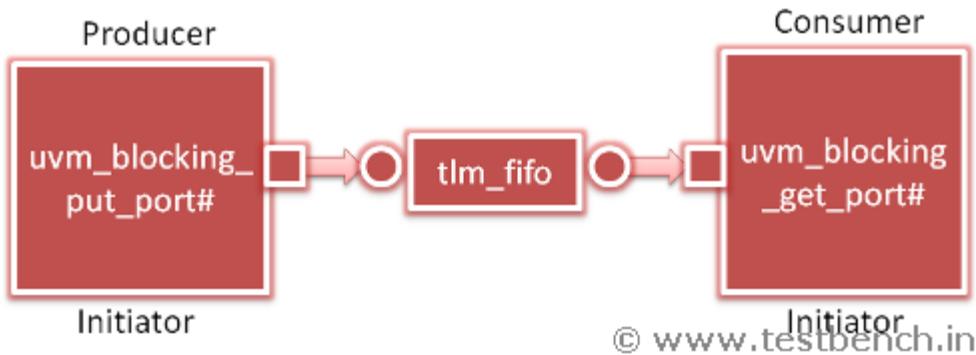
Removes all entries from the FIFO, after which used returns 0 and is\_empty returns 1.

### Example

Lets implement a example.

In this example, we will use a tlm\_fifo to connect producer and consumer.

The producer component generates the transaction and using its put\_port port() method, sends transaction out. The consumer component, to get the transaction from outside, uses get() method of get\_port. These two ports are connected to tlm\_fifo in the env class. In this example, producer and consumer are initiators as both components are calling the methods.



#### Producer source code:

```

class producer extends uvm_component;
  uvm_blocking_put_port #(int) put_port;
  function new(string name, uvm_component p = null);
    super.new(name,p);
    put_port = new("put_port", this);
  endfunction
  task run;
    int randval;
    for(int i = 0; i < 10; i++)
      begin
        #10;
        randval = $urandom_range(4,10);
        `uvm_info("producer",
          $sformatf("sending %d",randval), UVM_MEDIUM)
        put_port.put(randval);
      end
  endtask
endclass : producer

```

#### Consumer source code:

```

class consumer extends uvm_component;
  uvm_blocking_get_port #(int) get_port;
  function new(string name, uvm_component p = null);
    super.new(name,p);
    get_port = new("get_port", this);
  endfunction
  task run;
    int val;
    forever
      begin
        get_port.get(val);
      end
  endtask
endclass : consumer

```

```

`uvm_info("consumer",
    $sformatf("receiving %d", val), UVM_MEDIUM)
end
endtask
endclass : consumer

```

Env source code:

```

class env extends uvm_env;
producer p;
consumer c;
tlm_fifo #(int) f;
function new(string name = "env");
super.new(name);
p = new("producer", this);
c = new("consumer", this);
f = new("fifo", this);
endfunction
function void connect();
p.put_port.connect(f.put_export);
c.get_port.connect(f.get_export);
endfunction
task run();
#1000 global_stop_request();
endtask
endclass

```

Download the example

[uvm\\_tlm\\_4.tar](#)

[Browse the code in uvm\\_tlm\\_4.tar](#)

Command to sun the simulation

VCS Users : make vcs

Questa Users: make questa

Log

UVM_INFO producer.sv(28) @ 10: env.producer [producer] sending	7
UVM_INFO consumer.sv(26) @ 10: env.consumer [consumer] receiving	7
UVM_INFO producer.sv(28) @ 20: env.producer [producer] sending	4
UVM_INFO consumer.sv(26) @ 20: env.consumer [consumer] receiving	4

**UVM CALLBACK**

Callback mechanism is used for altering the behavior of the transactor without modifying the transactor. One of the many promises of Object-Oriented programming is that it will allow for plug-and-play re-usable verification components. Verification Designers will hook the transactors together to make a verification environment. In SystemVerilog, this hooking together

of transactors can be tricky. Callbacks provide a mechanism whereby independently developed objects may be connected together in simple steps.

This article describes uvm callbacks. uvm callback might be used for simple notification, two-way communication, or to distribute work in a process. Some requirements are often unpredictable when the transactor is first written. So a transactor should provide some kind of hooks for executing the code which is defined afterwards. In uvm, these hooks are created using callback methods. For instance, a driver is developed and an empty method is called before driving the transaction to the DUT. Initially this empty method does nothing. As the implementation goes, user may realize that he needs to print the state of the transaction or to delay the transaction driving to DUT or inject an error into transaction. Callback mechanism allows executing the user defined code in place of the empty callback method. Other example of callback usage is in monitor. Callbacks can be used in a monitor for collecting coverage information or for hooking up to scoreboard to pass transactions for self checking. With this, user is able to control the behavior of the transactor in verification environment and individual testcases without doing any modifications to the transactor itself.

Following are the steps to be followed to create a transactor with callbacks. We will see simple example of creating a Driver transactor to support callback mechanism.

Step 1) Define a facade class.

1) Extend the uvm\_callback class to create a faced class.

```
class Driver_callback extends uvm_callback;  
endclass : Driver_callback
```

2) Define required callback methods. All the callback methods must be virtual.

In this example, we will create callback methods which will be called before driving the packet and after driving the packet to DUT.

```
virtual task pre_send(); endtask  
virtual task post_send(); endtask
```

3) Define the constructor and get\_type\_name methods and define type\_name.

```
function new (string name = "Driver_callback");  
    super.new(name);  
endfunction  
static string type_name = "Driver_callback";  
virtual function string get_type_name();  
    return type_name;  
endfunction
```

Step 2) Register the facade class with driver and call the callback methods.

1) In the driver class, using `uvm\_register\_cb() macro, register the facade class.

```
class Driver extends uvm_component;  
`uvm_component_utils(Driver)  
`uvm_register_cb(Driver,Driver_callback)  
function new (string name, uvm_component parent=null);
```

```

super.new(name,parent);
endfunction
endclass

```

2)Calling callback method.

Inside the transactor, callback methods should be called whenever something interesting happens.

We will call the callback method before driving the packet and after driving the packet. We defined 2 methods in facade class. We will call pre\_send() method before sending the packet and post\_send() method after sending the packet.

Using a `uvm\_do\_callbacks() macro, callback methods are called.

There are 3 arguments to `uvm\_do\_callbacks() macro.

First argument must be the driver class and second argument is facade class.

Third argument must be the callback method in the facade class.

To call pre\_send() method , use macro

`uvm\_do\_callbacks(Driver,Driver\_callback,pre\_send());

and similarly to call post\_send() method,

`uvm\_do\_callbacks(Driver,Driver\_callback,post\_send());

Place the above macros before and after driving the packet.

```

virtual task run();
repeat(2) begin
    `uvm_do_callbacks(Driver,Driver_callback,pre_send())
    $display(" Driver: Started Driving the packet ..... %d",$time);
    // Logic to drive the packet goes here
    // let's consider that it takes 40 time units to drive a packet.
    #40;
    $display(" Driver: Finished Driving the packet ..... %d",$time);
    `uvm_do_callbacks(Driver,Driver_callback,post_send())
end
endtask

```

With this, the Driver implementation is completed with callback support.

### Driver And Driver Callback Class Source Code

```

class Driver_callback extends uvm_callback;
    function new (string name = "Driver_callback");
        super.new(name);
    endfunction
    static string type_name = "Driver_callback";

    virtual function string get_type_name();
        return type_name;
    endfunction

```

```

virtual task pre_send(); endtask
virtual task post_send(); endtask
endclass : Driver_callback
class Driver extends uvm_component;
`uvm_component_utils(Driver)

`uvm_register_cb(Driver,Driver_callback)
function new (string name, uvm_component parent=null);
    super.new(name,parent);
endfunction
virtual task run();
    repeat(2) begin
        `uvm_do_callbacks(Driver,Driver_callback,pre_send())
        $display(" Driver: Started Driving the packet ..... %d",$time);
        // Logic to drive the packet goes here
        // let's consider that it takes 40 time units to drive a packet.
        #40;
        $display(" Driver: Finished Driving the packet ..... %d",$time);
        `uvm_do_callbacks(Driver,Driver_callback,post_send())
    end
endtask
endclass

```

Let's run the driver in simple testcase. In this testcase, we are not changing any callback methods definitions.

### Testcase Source Code

```

module test;
Driver drvr;
initial begin
    drvr = new("drvr");
    run_test();
end
endmodule

```

### Download files

[uvm\\_callback\\_1.tar](#)

[Browse the code in uvm\\_callback\\_1.tar](#)

### Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

## Log report

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
Driver: Started Driving the packet ..... 0
Driver: Finished Driving the packet ..... 40
Driver: Started Driving the packet ..... 40
Driver: Finished Driving the packet ..... 80
```

UVM\_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.

Following steps are to be performed for using callback mechanism to do required functionality.  
We will see how to use the callbacks which are implemented in the above defined driver in a testcase.

- 1) Implement the user defined callback method by extending facade class of the driver class.  
We will delay the driving of packet be 20 time units using the pre\_send() call back method.  
We will just print a message from post\_send() callback method.

```
class Custom_Driver_callbacks_1 extends Driver_callback;
    function new (string name = "Driver_callback");
        super.new(name);
    endfunction
    virtual task pre_send();
        $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
        #20;
    endtask
    virtual task post_send();
        $display("CB_1:post_send: Just a message from post send callback method \n");
    endtask
endclass
```

- 2) Construct the user defined facade class object.

```
Custom_Driver_callbacks_1 cb_1;
cb_1 = new("cb_1");
```

- 3) Register the callback method with the driver component. uvm\_callback class has static method add() which is used to register the callback.

```
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);
```

## Testcase 2 Source Code

```
class Custom_Driver_callbacks_1 extends Driver_callback;
    function new (string name = "Driver_callback");
        super.new(name);
```

```

endfunction
virtual task pre_send();
    $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
    #20;
endtask
virtual task post_send();
    $display("CB_1:post_send: Just a message from post send callback method \n");
endtask
endclass
module test;
initial begin
    Driver drvr;
    Custom_Driver_callbacks_1 cb_1;
    drvr = new("drvr");
    cb_1 = new("cb_1");
    uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);
    uvm_callbacks #(Driver,Driver_callback)::display();
    run_test();
end
endmodule

```

### Download the example

[uvm\\_callback\\_2.tar](#)

[Browse the code in uvm\\_callback\\_2.tar](#)

### Simulation Command

VCS Users : make vcs

Questa Users: make questa

Run the testcase. See the log results; We delayed the driving of packet by 20 time units using callback mechanism. See the difference between the previous testcase log and this log.

### Log report

cb\_1 on drvr ON

UVM\_INFO @ 0: reporter [RNTST] Running test ...

CB_1:pre_send: Delaying the packet driving by 20 time units.	0
--	---

Driver: Started Driving the packet .....	20
--	----

Driver: Finished Driving the packet .....	60
---	----

CB\_1:post\_send: Just a message from post send callback method

CB_1:pre_send: Delaying the packet driving by 20 time units.	60
--	----

Driver: Started Driving the packet .....	80
--	----

Driver: Finished Driving the packet .....	120
---	-----

CB\_1:post\_send: Just a message from post send callback method

UVM\_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.

Now we will see registering 2 callback methods.

1) Define another user defined callback methods by extending facade class.

```
class Custom_Driver_callbacks_2 extends Driver_callback;
    function new (string name = "Driver_callback");
        super.new(name);
    endfunction
    virtual task pre_send();
        $display("CB_2:pre_send: Hai .... this is from Second callback %d",$time);
    endtask
endclass
```

2) Construct the user defined facade class object.

```
Custom_Driver_callbacks_2 cb_2;
cb_2 = new("cb_2");
```

3) Register the object

```
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);
```

### Testcase 3 Source Code

```
class Custom_Driver_callbacks_1 extends Driver_callback;
function new (string name = "Driver_callback");
    super.new(name);
endfunction
virtual task pre_send();
    $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
    #20;
endtask
virtual task post_send();
    $display("CB_1:post_send: Just a message from post send callback method \n");
endtask
endclass
class Custom_Driver_callbacks_2 extends Driver_callback;
    function new (string name = "Driver_callback");
        super.new(name);
    endfunction

    virtual task pre_send();
        $display("CB_2:pre_send: Hai .... this is from Second callback %d",$time);
    endtask
endclass
module test;
```

### **initial begin**

```
Driver drvr;  
Custom_Driver_callbacks_1 cb_1;  
Custom_Driver_callbacks_2 cb_2;  
drvr = new("drvr");  
cb_1 = new("cb_1");  
cb_2 = new("cb_2");  
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);  
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);  
uvm_callbacks #(Driver,Driver_callback)::display();  
run_test();
```

**end**

**endmodule**

[Download source code](#)

[uvm\\_callback\\_3.tar](#)

[Browse the code in uvm\\_callback\\_3.tar](#)

### Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

Run the testcase and analyze the result.

### Log report

```
UVM_INFO @ 0: reporter [RNTST] Running test ...  
CB_1:pre_send: Delaying the packet driving by 20 time units. 0  
CB_2:pre_send: Hai .... this is from Second callback 20  
Driver: Started Driving the packet ..... 20  
Driver: Finished Driving the packet ..... 60  
CB_1:post_send: Just a message from post send callback method  
  
CB_1:pre_send: Delaying the packet driving by 20 time units. 60  
CB_2:pre_send: Hai .... this is from Second callback 80  
Driver: Started Driving the packet ..... 80  
Driver: Finished Driving the packet ..... 120  
CB_1:post_send: Just a message from post send callback method  
UVM_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.
```

The log results show that pre\_send() method of CDc\_1 is called first and then pre\_send() method of Cdc\_2. This is because of the order of the registering callbacks.

`uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);`

```
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);
```

Now we will see how to change the order of the callback method calls.

By changing the sequence of calls to add() method, order of callback method calling can be changed.

#### **Testcase 4 Source Code**

```
module test;
```

#### **initial begin**

```
Driver drvr;  
Custom_Driver_callbacks_1 cb_1;  
Custom_Driver_callbacks_2 cb_2;  
drvr = new("drvr");  
cb_1 = new("cb_1");  
cb_2 = new("cb_2");  
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_2);  
uvm_callbacks #(Driver,Driver_callback)::add(drvr,cb_1);  
uvm_callbacks #(Driver,Driver_callback)::display();  
run_test();
```

```
end
```

```
endmodule
```

#### **Download the source code**

[uvm\\_callback\\_4.tar](#)

[Browse the code in uvm\\_callback\\_4.tar](#)

#### **Command to run the simulation**

VCS Users : make vcs

Questa Users: make questa

Run and analyze the results.

Log results show that, pre\_send() method of CDs\_1 is called after calling CDs\_2 pre\_send() method.

#### **Log file report**

```
UVM_INFO @ 0: reporter [RNTST] Running test ...
```

```
CB_2:pre_send: Hai .... this is from Second callback 0
```

```
CB_1:pre_send: Delaying the packet driving by 20 time units. 0
```

```
Driver: Started Driving the packet ..... 20
```

```
Driver: Finished Driving the packet ..... 60
```

```
CB_1:post_send: Just a message from post send callback method
```

```
CB_2:pre_send: Hai .... this is from Second callback 60
```

```
CB_1:pre_send: Delaying the packet driving by 20 time units. 60
```

```
Driver: Started Driving the packet ..... 80
```

```
Driver: Finished Driving the packet ..... 120
```

CB\_1:post\_send: Just a message from post send callback method  
UVM\_ERROR @ 9200: reporter [TIMOUT] Watchdog timeout of '9200' expired.

### **Methods:**

#### **add\_by\_name:**

We have seen, the usage of add() method which requires object.

Using add\_by\_name() method, callback can be registered with object name.

**static function void add\_by\_name(string name,**

```
    uvm_callback cb,  
    uvm_component root,  
    uvm_appprepend ordering = UVM_APPEND)
```

#### **delete:**

uvm also provides uvm\_callbacks::delete() method to remove the callback methods which are registered.

Similar to delete, delete\_by\_name() method is used to remove the callback using the object name.

**static function void delete\_by\_name(string name,**

```
    uvm_callback cb,  
    uvm_component root )
```

### **Macros:**

**`uvm\_register\_cb**

Registers the given CB callback type with the given T object type.

**`uvm\_set\_super\_type**

Defines the super type of T to be ST.

**`uvm\_do\_callbacks**

Calls the given METHOD of all callbacks of type CB registered with the calling object

**`uvm\_do\_obj\_callbacks**

Calls the given METHOD of all callbacks based on type CB registered with the given object, OBJ, which is or is based on type T.

**`uvm\_do\_callbacks\_exit\_on**

Calls the given METHOD of all callbacks of type CB registered with the calling object

**`uvm\_do\_obj\_callbacks\_exit\_on**

Calls the given METHOD of all callbacks of type CB registered with the given object OBJ, which must be or be based on type T, and returns upon the first callback that returns the bit value given by VAL.

## **INTRODUCTION**

Verification methodological manual (VMM) , co-authored by verification experts from ARM and Synopsys, describes how to use SystemVerilog to develop scalable, predictable and reusable verification environments. VMM has become important factor in increasing verification reuse, improved verification productivity and timeliness.

VMM consists coding guide lines and base classes. VMM is focused on Coverage driven verification methodology. VMM supports both the top-down and bottom-up approaches. VMM follows layered test bench architecture to take the full advantage of the automation. The VMM for SystemVerilog TestBench architecture comprises five layers.

The layered TestBench is the heart of the verification environment in VMM:

**⌚ signal layer:**

This layer connects the TestBench to the RTL design. It consists of interface, clocking, and modport constructs.

**⌚ command layer:**

This layer contains lower-level driver and monitor components, as well as the assertions. This layer provides a transaction-level interface to the layer above and drives the physical pins via the signal layer.

**⌚ functional layer:**

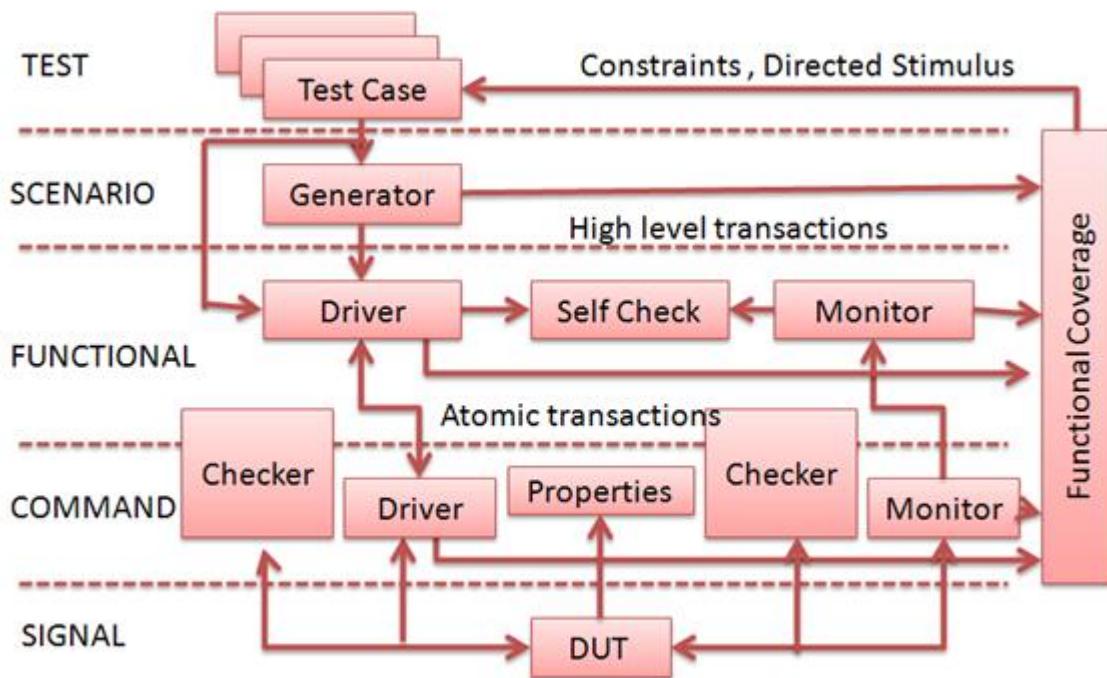
This layer contains higher-level driver and monitor components, as well as the self-checking structure (scoreboard/tracker).

**⌚ scenario layer:**

This layer uses generators to produce streams or sequences of transactions that are applied to the functional layer. The generators have a set of weights, constraints or scenarios specified by the test layer. The randomness of constrained-random testing is introduced within this layer.

**⌚ test layer:**

Tests are located in this layer. Test layer can interact with all the layers. This layer allows to pass directed commands to functional and command layer.



© [www.testbench.in](http://www.testbench.in)

VMM libraries consists following sub libraries

- VMM Standard Library
- VMM Register Abstraction Layer (RAL)
- VMM Hardware Abstraction Layer (HAL)
- VMM Scoreboarding

The VMM Standard Library provides base classes for key aspects of the verification environment, transaction generation, notification service and a message logging service.

These libraries can be downloaded from <http://www.vmmcentral.org>

Following are some of the classes and macros defined in the VMM Standard Library

**• vmm\_env :**

The class is a base class used to implement verification environments.

**• vmm\_xactor :**

This base class is to be used as the basis for all transactors, including bus-functional models, monitors and generators. It provides a standard control mechanism expected to be found in all transactors.

**• vmm\_channel :**

This class implements a generic transaction-level interface mechanism. Transaction-level interfaces remove the higher-level layers from the physical interface details. Using channels, transactors pass transactions from one to other.

### **⌚ vmm\_data :**

This base class is to be used as the basis for all transaction descriptors and data models. It provides a standard set of methods expected to be found in all descriptors. User must extend vmm\_data to create a custom transaction.

### **⌚ vmm\_log :**

The vmm\_log class used implements an interface to the message service. These classes provide a mechanism for reporting simulation activity to a file or a terminal. To ensure a consistent look and feel to the messages issued from different sources, vmm\_log is used.

### **⌚ vmm\_atomic\_gen :**

This is a macro. This macro defines a atomic generator for generating transaction which are derived from vmm\_data.

### **⌚ vmm\_scenario\_gen :**

Defines a scenario generator class to generate sequences of related instances of the specified class.

### **⌚ vmm\_notify :**

The vmm\_notify class implements an interface to the notification service. The notification service provides a synchronization mechanism for concurrent threads or transactors.

### **⌚ vmm\_test :**

This class will be useful for runtime selection of testcases to run on an environment.

## **VMM LOG**

The vmm\_log class provides an interface to the VMM message service so that all messages, regardless of their sources, can have a common "look and feel". Not always we require all type of messages. During normal simulation, we need only note, error and warning messages. While you are debugging, you may need debug messages and trace messages. vmm\_log allows you to control the messages. This helps in debugging. vmm\_log has a uniform message format which helps for post processing the log file using scripts if needed. You can also convert a error message to warning message which is required in some specific testcases. The message service describes and controls messages based on several concepts:

### **⌚ Message source:**

A message source can be any component of a TestBench. They can be Transactors, scoreboards, assertions, environment or a testcase.

### **⌚ Message filters:**

Filters can prevent or allow a message from being issued. They can be promoted or demoted based on the identifier, type, severity or content.

## **Vmm Message Type**

Individual messages are categorized into different types.

### **⌚ FAILURE\_TYP : For reporting error messages.**

### **⌚ NOTE\_TYP : Normal message used to indicate the simulation progress.**

- ⌚ DEBUG\_TYP : Message useful for debugging purpose.
- ⌚ TIMING\_TYP : For reporting timing errors.
- ⌚ XHANDLING\_TYP: For reporting X or Z on signals
- ⌚ INTERNAL\_TYP : Messages from the VMM base classes.
- ⌚ REPORT\_TYP,PROTOCOL\_TYP,TRANSACTION\_TYP,COMMAND\_TYP,CYCLE\_TYP
- :⌚ Additional message types that can be used by transactors.

### Message Severity

Individual messages are categorized into different severities

- ⌚ FATAL\_SEV : An error which causes a program to abort.
- ⌚ ERROR\_SEV : Simulation aborts after a certain number of errors are observed.
- ⌚ WARNING\_SEV: Simulation can proceed and still produce useful result.
- ⌚ NORMAL\_SEV : This message indicates the state of simulation.
- ⌚ TRACE\_SEV : This message identifies high-level internal information that is not normally issued.
- ⌚ DEBUG\_SEV : This message identifies medium-level internal information that is not normally issued.
- ⌚ VERBOSE\_SEV: This message identifies low-level internal information that is not normally issued.

### Vmm Log Macros

We can use the following predefined macros to select the message severity.

```
`vmm_fatal(vmm_log log, string msg);
`vmm_error(vmm_log log, string msg);
`vmm_warning(vmm_log log, string msg);
`vmm_note(vmm_log log, string msg);
`vmm_trace(vmm_log log, string msg);
`vmm_debug(vmm_log log, string msg);
`vmm_verbose(vmm_log log, string msg);
```

The second argument in the above macro can accept only a string. So if we want to pass a complex message, then convert it to string using a \$psprintf() system task.

These macros are simple to use and the macro expansion is easy to understand. Following is a expansion of `vmm\_note macro. Other macros also have same logic with different Message severity levels.

```
`define vmm_note ( log, msg )
do
  if (log.start_msg(vmm_log::NOTE_TYP)) begin
    void'(log.text(msg));
    log.end_msg();
  end
  while (0)
```

To change the severity and verbosity of the messages services at simulation time , use +vmm\_log\_default=SEVERITY TYPE. Where SEVERITY TYPE can be ERROR, WARNING, NORMAL, TRACE, DEBUG OR VERBOSE. We will see the demo of this service in the example.

### **Message Handling**

Different messages require different action by the simulator once the message has been issued.

- ⌚ **ABORT\_SIM** : Aborts the simulation.
- ⌚ **COUNT\_ERROR**: Count the message as an error.
- ⌚ **STOP\_PROMPT**: Stop the simulation immediately and return to the simulation runtime-control command prompt.
- ⌚ **DEBUGGER** : Stop the simulation immediately and start the graphical debugging environment.
- ⌚ **DUMP\_STACK** : Dump the call stack or any other context status information and continue the simulation.
- ⌚ **CONTINUE** : Continue the simulation normally.

To use this vmm feature, take the instance of vmm\_log and use the message macros. For most of the vmm classes like vmm\_env,vmm\_xactor,vmm\_data etc, this instantiation is already done internally, so user dont need to take a separate instance of vmm\_log.

```
vmm_log log = new("test_log","log");
```

Following is a simple program which demonstrates the usages of different severity levels. This program creates the object of a vmm\_log class and uses log macros to define various messages. You can also see the usage of \$psprintf() system task.

```
program test_log();
vmm_log log = new("test_log","log");
initial begin
    `vmm_error(log,"This is a ERROR Message");
    `vmm_warning(log,"This is a WARNING Message");
    `vmm_note(log,$psprintf("This is a NOTE Message at time %d",$time));
    `vmm_trace(log,"This is a TRACE Message");
    `vmm_debug(log,"This is a DEBUG Message");
    `vmm_verbose(log,"This is a VERBOSE Message");
    `vmm_fatal(log,"This is a FATAL Message");
end
endprogram
```

[Download the source code](#)

[vmm\\_log.tar](#)

[Browse the code in vmm\\_log.tar](#)

### Simulation commands

```
vcs -sverilog -f filelist -R -ntb_opts rvm -ntb_opts dtm
```

You can see the following messages on the terminal. You can only see Normal, Error, warning and fatal messages.

### Log file report:

```
!ERROR![FAILURE] on test_log(log) at 0:
```

This is a ERROR Message

```
WARNING[FAILURE] on test_log(log) at 0:
```

This is a WARNING Message

```
Normal[NOTE] on test_log(log) at 0:
```

This is a NOTE Message at time 0

```
*FATAL*[FAILURE] on test_log(log) at 0:
```

This is a FATAL Message

Run the simulation with command option +vmm\_log\_default=DEBUG

You can see the following messages on the terminal. You can see that Debug and trace message is also coming.

### Log file report:

```
!ERROR![FAILURE] on test_log(log) at 0:
```

This is a ERROR Message

```
WARNING[FAILURE] on test_log(log) at 0:
```

This is a WARNING Message

```
Normal[NOTE] on test_log(log) at 0:
```

This is a NOTE Message at time 0

```
Trace[DEBUG] on test_log(log) at 0:
```

This is a TRACE Message

```
Debug[DEBUG] on test_log(log) at 0:
```

This is a DEBUG Message

```
*FATAL*[FAILURE] on test_log(log) at 0:
```

This is a FATAL Message

Run the simulation with command option +vmm\_log\_default=VERBOSE

You can see the following messages on the terminal. You can see that Trace, Debug and Verbose messages are also coming.

## Log file report:

```
!ERROR![FAILURE] on test_log(log) at          0:  
    This is a ERROR Message  
WARNING[FAILURE] on test_log(log) at          0:  
    This is a WARNING Message  
Normal[NOTE] on test_log(log) at             0:  
    This is a NOTE Message at time            0  
Trace[DEBUG] on test_log(log) at             0:  
    This is a TRACE Message  
Debug[DEBUG] on test_log(log) at             0:  
    This is a DEBUG Message  
Verbose[DEBUG] on test_log(log) at           0:  
    This is a VERBOSE Message  
*FATAL*[FAILURE] on test_log(log) at         0:  
    This is a FATAL Message
```

## Counting Number Of Messages Based Of Message Severity

Some time we need to count the number of messages executed based of the severity type.  
vmm\_log has get\_message\_count() function which returns the message count based on severity type, source and message string.  
user dont need to implement a logic to print the state of test i.e TEST PASSED or TEST FAILED, vmm\_env has already implemented this in the report() method. We will see this in next topic.

```
virtual function int get_message_count(int severity = ALL_SEVS,  
                                      string name = "",  
                                      string instance = "",  
                                      bit recurse = 0);
```

Following is a example which demonstrates the get\_message\_count() function. The following example , also demonstrates the use of \$psprintf which will be use full to print message when there are variable arguments to print.

```
program test_log();  
vmm_log log = new("test_log","log");  
int fatal_cnt ;  
int error_cnt ;  
int warn_cnt ;  
initial begin  
    `vmm_note(log,$psprintf("This is a NOTE Message at time %d",$time));  
  
    `vmm_error(log,"This is a ERROR Message 1 ");  
    `vmm_error(log,"This is a ERROR Message 2 ");
```

```

`vmm_error(log,"This is a ERROR Message 3 ");
`vmm_warning(log,"This is a WARNING Message 1 ");
    `vmm_warning(log,"This is a WARNING Message 2 ");

fatal_cnt = log.get_message_count(vmm_log::FATAL_SEV, "./", "./", 1);
error_cnt = log.get_message_count(vmm_log::ERROR_SEV, "./", "./", 1);
warn_cnt = log.get_message_count(vmm_log::WARNING_SEV, "./", "./", 1);
$display("\n\n");
$display(" Number of Fatal messages : %0d ",fatal_cnt);
$display(" Number of Error messages : %0d ",error_cnt);
$display(" Number of Warning messages : %0d ",warn_cnt);
$display("\n\n");
end

```

### **endprogram**

#### Download the source code

[vmm\\_log\\_1.tar](#)

[Browse the code in vmm\\_log\\_1.tar](#)

#### Simulation commands

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

#### Log file report:

```

Normal[NOTE] on test_log(log) at      0:
  This is a NOTE Message at time   0
!ERROR![FAILURE] on test_log(log) at 0:
  This is a ERROR Message 1
!ERROR![FAILURE] on test_log(log) at 0:
  This is a ERROR Message 2
!ERROR![FAILURE] on test_log(log) at 0:
  This is a ERROR Message 3
WARNING[FAILURE] on test_log(log) at 0:
  This is a WARNING Message 1
WARNING[FAILURE] on test_log(log) at 0:
  This is a WARNING Message 2
Number of Fatal messages : 0
Number of Error messages : 3
Number of Warning messages : 2

```

Sometimes we would like to wait for a specific message and execute some logic. For example, you want to count number of crc error packets transmitted by dut in some special testcase or you want to stop the simulation after a specific series of messages are received. vmm\_log has a method to wait for a specific message.

```

virtual task wait_for_msg(string name = "",  

    string instance = "",  

    bit recurse = 0,  

    int typs = ALL_TYPs,  

    int severity = ALL_SEVs,  

    string text = "",  

    logic issued = 1'bx,  

    ref vmm_log_msg msg);

```

Following is a simple example which demonstrates the use of above method. In this example, wait\_for\_msg() method wait for a specific string.

```
program test_log();
```

```

vmm_log log = new("test_log","log");  

vmm_log_msg msg = new(log);  

/* This logic is some where in the monitor */  

initial  

repeat (4) begin  

#($urandom()%10)  

`vmm_error(log,"Packet with CRC ERROR is received");  

end  

/* In testcase you are counting the error messages */  

initial  

forever begin  

log.wait_for_msg("./","./",-1,vmm_log::ALL_TYPs,vmm_log::ERROR_SEV,"Packet with  

CRC ERROR is received",1'bx,msg);  

// You can count number of crc errored pkts rcvd.  

// or do whatever you want.  

$display(" -- Rcvd CRC ERROR message at %d -- ",$time);  

end  

endprogram

```

[Download the source code](#)

[vmm\\_log\\_2.tar](#)

[Browse the code in vmm\\_log\\_2.tar](#)

[Simulation commands](#)

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

[file report:](#)

```

!ERROR![FAILURE] on test_log(log) at          8:  

  Packet with CRC ERROR is received  

-- Rcvd CRC ERROR message at          8 --  

!ERROR![FAILURE] on test_log(log) at          8:  


```

```

Packet with CRC ERROR is received
-- Rcvd CRC ERROR message at      8 --
!ERROR![FAILURE] on test_log(log) at      11:
Packet with CRC ERROR is received
-- Rcvd CRC ERROR message at      11 --
!ERROR![FAILURE] on test_log(log) at      11:
Packet with CRC ERROR is received
-- Rcvd CRC ERROR message at      11 -

```

## **VMM ENV**

Verification environment is developed by extending vmm\_env class. The TestBench simulation needs some systematic flow like reset, initialize etc. vmm\_env base class has methods formalize the simulation steps. All methods are declared as virtual methods. All the Verification components and instantiated, connected and component activities starting is done in this class. As it contains all verification components instances, the environment class affects the entire test environment.

The vmm\_env class divides a simulation into the following steps, with corresponding methods:

**⌚ gen\_cfg()** :

Randomize test configuration parameters.

**⌚ build()** :

Creates the instances of channels and transactors. Transactors are connected using channels. DUT is also connected to TestBench using the interfaces.

**⌚ reset\_dut()** :

Reset the DUT using the interface signals.

**⌚ cfg\_dut()** :

Configures the DUT configuration parameters.

**⌚ start()** :

Starts all the components of the verification environment to start component activity. All the transactors (atomic\_gen, scenario\_gen, vmm\_xactor....) have start method, which starts their activities. All the components start() methods are called in this method.

**⌚ wait\_for\_end()** :

This method waits till the test is done.

**⌚ stop()** :

Stops all the components of the verification environment to terminate the simulation cleanly. Stop data generators.

⌚ cleanup() :

Performs clean-up operations to let the simulation terminate gracefully. It waits for DUT to drain all the data it has.

⌚ report() :

The report method in vmm\_env collects error and warning metrics from all the log objects and reports a summary of the results.

To create a user environment, define a new class extended from vmm\_env and extend the above methods. To retain the core functionality of the base class methods, each extended method must call super. as the first line of code.

In The following example, we are defining a new class Custom\_env from vmm\_env and extending all the vmm\_env class methods. All the methods are extended and `vmm\_note() message is included to understand the simulation flow.

```
class Custom_env extends vmm_env;

function new();
    super.new("Custom_env");
endfunction

virtual function void gen_cfg();
    super.gen_cfg();
    `vmm_note(this.log,"Start of gen_cfg() method ");
    `vmm_note(this.log,"End of gen_cfg() method ");
endfunction

virtual function void build();
    super.build();
    `vmm_note(this.log,"Start of build() method ");
    `vmm_note(this.log,"End of build() method ");
endfunction

virtual task reset_dut();
    super.reset_dut();
    `vmm_note(this.log,"Start of reset_dut() method ");
    `vmm_note(this.log,"End of reset_dut() method ");
endtask
```

```
virtual task cfg_dut();
  super.cfg_dut();
  `vmm_note(this.log,"Start of cfg_dut() method ");
  `vmm_note(this.log,"End of cfg_dut() method ");
endtask

virtual task start();
  super.start();
  `vmm_note(this.log,"Start of start() method ");
  `vmm_note(this.log,"End of start() method ");
endtask

virtual task wait_for_end();
  super.wait_for_end();
  `vmm_note(this.log,"Start of wait_for_end() method ");
  `vmm_note(this.log,"End of wait_for_end() method ");
endtask

virtual task stop();
  super.stop();
  `vmm_note(this.log,"Start of stop() method ");
  `vmm_note(this.log,"End of stop() method ");
endtask

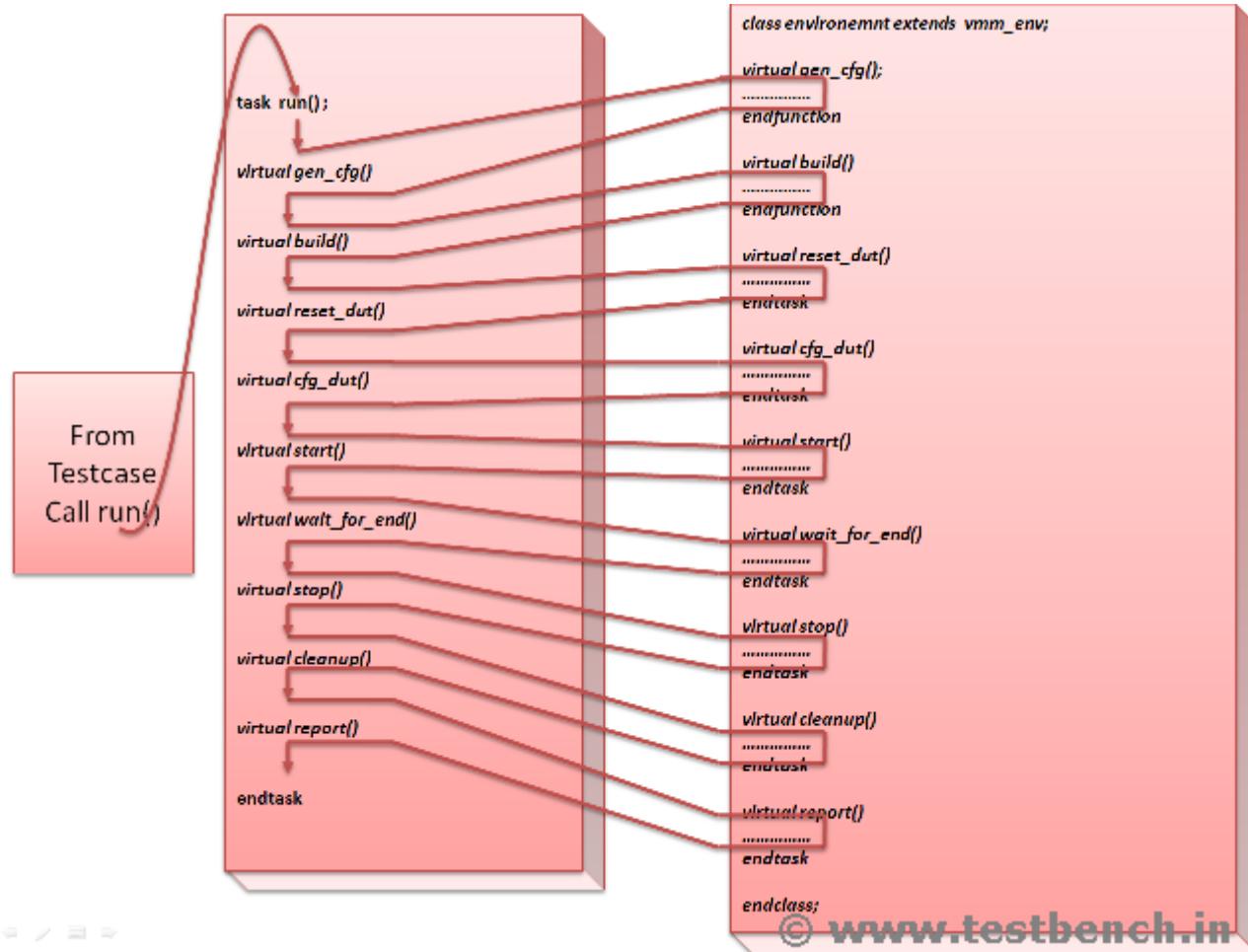
virtual task cleanup();
  super.cleanup();
  `vmm_note(this.log,"Start of cleanup() method ");
  `vmm_note(this.log,"End of cleanup() method ");
endtask

virtual task report();
  super.report();
  `vmm_note(this.log,"Start of report() method \n\n\n");
  `vmm_note(this.log,"End of report() method");
endtask

endclass
```

In addition to the methods that have already been discussed earlier, vmm\_env also contains a run() method which does not require any user extension. This method is called from the testcase. When this method is called, individual steps in vmm\_env are called in a sequence in the following order:

gen\_cfg => build => cfg\_dut\_t => start\_t => wait\_for\_end\_t => stop\_t => cleanup\_t => report



Now we will see the testcase implementation. VMM recommends the TestBench to be implemented in program block. In a program block, create an object of the Custom\_env class and call the run() method.

```

program test();
    Custom_env env = new();
    initial
        env.run();
    endprogram

```

## Download the files

[vmm\\_env\\_1.tar](#)

[Browse the code in vmm\\_env\\_1.tar](#)

### Simulation commands

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

### Log file report:

Normal[NOTE] on Custom_env() at	0:
Start of gen_cfg() method	
Normal[NOTE] on Custom_env() at	0:
End of gen_cfg() method	
Normal[NOTE] on Custom_env() at	0:
Start of build() method	
Normal[NOTE] on Custom_env() at	0:
End of build() method	
Normal[NOTE] on Custom_env() at	0:
Start of reset_dut() method	
Normal[NOTE] on Custom_env() at	0:
End of reset_dut() method	
Normal[NOTE] on Custom_env() at	0:
Start of cfg_dut() method	
Normal[NOTE] on Custom_env() at	0:
End of cfg_dut() method	
Normal[NOTE] on Custom_env() at	0:
Start of start() method	
Normal[NOTE] on Custom_env() at	0:
End of start() method	
Normal[NOTE] on Custom_env() at	0:
Start of wait_for_end() method	
Normal[NOTE] on Custom_env() at	0:
End of wait_for_end() method	
Normal[NOTE] on Custom_env() at	0:
Start of stop() method	
Normal[NOTE] on Custom_env() at	0:
End of stop() method	
Normal[NOTE] on Custom_env() at	0:
Start of cleanup() method	
Normal[NOTE] on Custom_env() at	0:
End of cleanup() method	
Simulation PASSED on ./ (./) at	0 (0 warnings, 0 demoted errors & 0 demoted

```

warnings)
Normal[NOTE] on Custom_env() at          0:
  Start of report() method
Normal[NOTE] on Custom_env() at          0:
  End of report() method
$finish at simulation time              0

```

Log file report shows that individual methods in vmm\_env are called in a ordered sequence upon calling run() method.

When you call build() and if gen\_cfg() is not called before that, gen\_cfg() will be called first then build will execute.

Same way, if you call gen\_cfg() followed by cfg\_dut() followed by run(), then cfg\_dut() will make sure to call build() followed by reset\_dut() first before executing its user defined logic, run() will make sure to call start(), wait\_for\_end(), stop(), clean(), report()in the order given.

### **program test();**

```

vmm_env env = new();
initial
begin
  $display("***** Before Calling env.gen_cfg() *****");
  env.gen_cfg();
  $display("***** Before Calling env.cfg_dut() *****");
  env.cfg_dut();
  $display("***** Before Calling env.run() *****");
  env.run();
end

```

### **endprogram**

#### Download the files

[vmm\\_env\\_2.tar](#)

[Browse the code in vmm\\_env\\_2.tar](#)

#### Simulation commands

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm +vmm\_log\_default=VERBOSE

#### Log file report:

```

***** Before Calling env.gen_cfg() *****
Trace[INTERNAL] on Verif Env() at      0:
  Generating test configuration...
***** Before Calling env.cfg_dut() *****
Trace[INTERNAL] on Verif Env() at      0:
  Building verification environment...
Trace[INTERNAL] on Verif Env() at      0:
  Reseting DUT...

```

```

Trace[INTERNAL] on Verif Env() at      0:
  Configuring...
***** Before Calling env.run() *****

Trace[INTERNAL] on Verif Env() at      0:
  Starting verification environment...
Trace[INTERNAL] on Verif Env() at      0:
  Saving RNG state information...
Trace[INTERNAL] on Verif Env() at      0:
  Waiting for end of test...
Trace[INTERNAL] on Verif Env() at      0:
  Stopping verification environment...
Trace[INTERNAL] on Verif Env() at      0:
  Cleaning up...
Simulation PASSED on ./ (./) at      0 (0 warnings, 0 demoted errors & 0 demoted
warnings)

```

## **VMM DATA**

vmm\_data class is to be used to model all transactions in the infrastructure . It provides a standard set of methods expected to be found in all transactions. All transactions in the verification environment inherit this object and override its main generic virtual tasks such as copy(), byte\_pack(), byte\_unpack(), compare() and psdisplay().

vmm\_data has 3 unique identifiers for identifying transaction instance.

```

int stream_id; // Stream identifier
int scenario_id;// Sequence identifier within stream
int data_id; // instance identifier within sequence

```

This class is used to generate random, constraint random and directed transactions.

We will see the implementation of some methods.

Let us implement a simple packet using vmm\_data.

Packet Specification

---

Packet has DA,SA,Len and ..... few more feilds.

All the feilds are 8 bit.

1) Define a packet class by extending vmm\_data

```
class Packet extends vmm_data;
```

2) Define all the packet fields as rand variables

```
rand bit [7:0] da;
rand bit [7:0] sa;
rand bit [7:0] length;
.....
....
```

3) Constraint the rand variables based on the specification.

```
constraint address_c { da inside {'P0,'P1,'P2,'P3} ; }
.....
.....
```

4) Define psdisplay() method.

psdisplay() method displays the current value of the transaction or data described by this instance in a human-readable format on the standard output.

```
virtual function string psdisplay(string prefix = "");
    int i;
    $write(psdisplay, " %s%s da:0x%h\n", psdisplay, prefix, this.da);
    $write(psdisplay, " %s%s sa:0x%h\n", psdisplay, prefix, this.sa);
    $write(psdisplay, " %s%s length:0x%h
(data.size=%0d)\n", psdisplay, prefix, this.length, this.data.size());
.....
endfunction
```

5) Define copy() method.

copy() method copies the current value of the object instance to the specified object instance.

```
virtual function vmm_data copy(vmm_data to = null);
    Packet cpy;
    cpy = new;
    super.copy_data(cpy);
    cpy.da = this.da;
    cpy.sa = this.sa;
    cpy.length = this.length;
.....
```

6) Define compare() method.

Compare method compares the current value of the object instance with the current value of the specified object instance.

```
virtual function bit compare(input vmm_data to, output string diff, input int kind = -1);
    Packet cmp;
    compare = 1; // Assume success by default.
```

```

diff = "No differences found";
// data types are the same, do comparison:
if (this.da != cmp.da)
begin
    diff = $psprintf("Different DA values: %b != %b", this.da, cmp.da);
    compare = 0;
end
if (this.sa != cmp.sa)
begin
    diff = $psprintf("Different SA values: %b != %b", this.sa, cmp.sa);
    compare = 0;
end
.....
.....

```

7) Define byte\_pack() method.

byte\_pack() method Packs the content of the transaction or data into the specified dynamic array of bytes.

```

virtual function int unsigned byte_pack(
    ref logic [7:0] bytes[],
    input int unsigned offset = 0 ,
    input int kind = -1);
byte_pack = 0;
bytes = new[this.data.size() + 4];
bytes[0] = this.da;
bytes[1] = this.sa;
bytes[2] = this.length;
.....
.....

```

8) Define byte\_unpack() method.

byte\_unpack() method unpacket the array in to different data feilds.

```

virtual function int unsigned byte_unpack(
    const ref logic [7:0] bytes[],
    input int unsigned offset = 0,
    input int len = -1,
    input int kind = -1);
this.da = bytes[0];
this.sa = bytes[1];
this.length = bytes[2];
.....

```

.....

### Complete Packet Class

```
class Packet extends vmm_data;
static vmm_log log = new("Packet","Class");
rand bit [7:0] length;
rand bit [7:0] da;
rand bit [7:0] sa;
rand bit [7:0] data[];//Payload using Dynamic array,size is generated on the fly
rand byte fcs;
constraint payload_size_c { data.size inside { [1 : 6]};}
function new();
    super.new(this.log);
endfunction:new
virtual function vmm_data allocate();
    Packet pkt;
    pkt = new();
    return pkt;
endfunction:allocate
virtual function string psdisplay(string prefix = "");
    int i;
    $write(psdisplay, " %s packet
#%0d.%0d.%0d\n", prefix,this.stream_id, this.scenario_id, this.data_id);
    $write(psdisplay, " %s%s da:0x%h\n", psdisplay, prefix,this.da);
    $write(psdisplay, " %s%s sa:0x%h\n", psdisplay, prefix,this.sa);
    $write(psdisplay, " %s%s length:0x%h
(data.size=%0d)\n", psdisplay, prefix,this.length,this.data.size());
    $write(psdisplay, " %s%s data[%0d]:0x%h", psdisplay, prefix,0,data[0]);
    if(data.size() > 1)
        $write(psdisplay, " data[%0d]:0x%h", 1,data[1]);
    if(data.size() > 4)
        $write(psdisplay, " .... ");
    if(data.size() > 2)
        $write(psdisplay, " data[%0d]:0x%h", data.size() -2,data[data.size() -2]);
    if(data.size() > 3)
        $write(psdisplay, " data[%0d]:0x%h", data.size() -1,data[data.size() -1]);
    $write(psdisplay, "\n %s%s fcs:0x%h \n", psdisplay, prefix, this.fcs);

endfunction
virtual function vmm_data copy(vmm_data to = null);
    Packet cpy;
```

```

// Copying to a new instance?
if (to == null)
    cpy = new;
else
// Copying to an existing instance. Correct type?
if (!$cast(cpy, to))
begin
`vmm_fatal(this.log, "Attempting to copy to a non packet instance");
copy = null;
return copy;
end
super.copy_data(cpy);
cpy.da = this.da;
cpy.sa = this.sa;
cpy.length = this.length;
cpy.data = new[this.data.size()];
foreach(data[i])
begin
    cpy.data[i] = this.data[i];
end
cpy.fcs = this.fcs;
copy = cpy;
endfunction:copy
virtual function bit compare(input vmm_data to,output string diff,input int kind= -1);
    Packet cmp;
    compare = 1; // Assume success by default.
    diff = "No differences found";
    if (!$cast(cmp, to))
begin
`vmm_fatal(this.log, "Attempting to compare to a non packet instance");
    compare = 0;
    diff = "Cannot compare non packets";
    return compare;
end
// data types are the same, do comparison:
if (this.da != cmp.da)
begin
    diff = $psprintf("Different DA values: %b != %b", this.da, cmp.da);
    compare = 0;
    return compare;

```

```

end
if (this.sa != cmp.sa)
begin
    diff = $psprintf("Different SA values: %b != %b", this.sa, cmp.sa);
    compare = 0;
    return compare;
end
if (this.length != cmp.length)
begin
    diff = $psprintf("Different LEN values: %b != %b", this.length, cmp.length);
    compare = 0;
    return compare;
end
foreach(data[i])
    if (this.data[i] != cmp.data[i])
        begin
            diff = $psprintf("Different data[%0d] values: 0x%h != 0x%h", i, this.data[i], cmp.data[i]);
            compare = 0;
            return compare;
        end
    if (this.fcs != cmp.fcs)
        begin
            diff = $psprintf("Different FCS values: %b != %b", this.fcs, cmp.fcs);
            compare = 0;
            return compare;
        end
endfunction:compare
virtual function int unsigned byte_pack(
    ref logic [7:0] bytes[],
    input int unsigned offset = 0 ,
    input int kind = -1);
byte_pack = 0;
bytes = new[this.data.size() + 4];
bytes[0] = this.da;
bytes[1] = this.sa;
bytes[2] = this.length;

foreach(data[i])
    bytes[3+i] = data[i];

```

```

bytes[this.data.size() + 3] = fcs;
byte_pack = this.data.size() + 4;
endfunction:byte_pack
virtual function int unsigned byte_unpack(
    const ref logic [7:0] bytes[],
    input int unsigned offset = 0,
    input int len = -1,
    input int kind = -1);
this.da = bytes[0];
this.sa = bytes[1];
this.length = bytes[2];
this.fcs = bytes[bytes.size() - 1];
this.data = new[bytes.size() - 4];
foreach(data[i])
    this.data[i] = bytes[i+3];
return bytes.size();
endfunction:byte_unpack
endclass

```

### Vmm\_data Methods

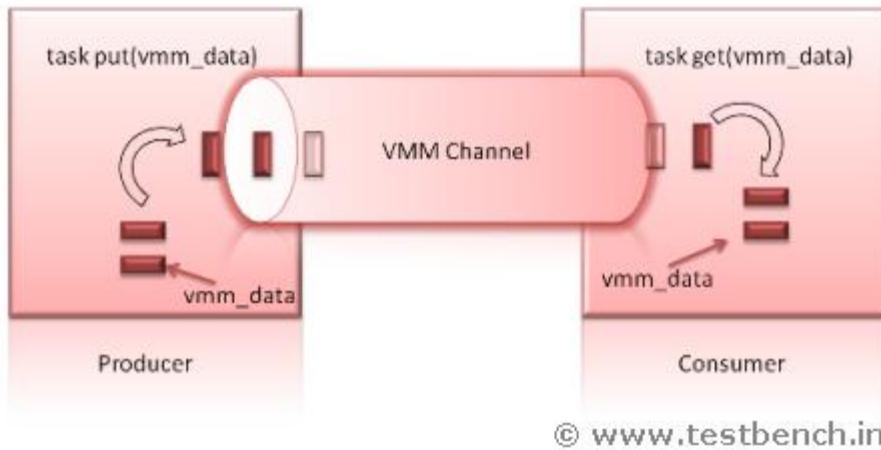
```

virtual function string psdisplay ( string prefix = "" );
virtual function bit is_valid ( bit silent = 1, int kind = -1 );
virtual function vmm_data allocate ( );
virtual function vmm_data copy ( vmm_data to = null );
virtual function bit compare (
    input vmm_data to, output string diff, input int kind = -1 );
function void display(string prefix = "");
virtual protected function void copy_data ( vmm_data to );
virtual function int unsigned byte_pack (
    ref logic [7:0] bytes [ ], int unsigned offset = 0, int kind = -1 );
virtual function int unsigned byte_unpack (
    const ref logic [7:0] bytes [ ], input int unsigned offset = 0,
    input int len = -1, input int kind = -1 );
virtual function int unsigned byte_size ( int kind = -1 );
virtual function int unsigned max_byte_size ( int kind = -1 );
virtual function void save ( int file );
virtual function bit load ( int file );

```

## **VMM CHANNEL**

The channel is the interface mechanism used by transactors to transfer transactions. Transaction objects are produced or consumed by a transactor. Transactor can be a generator or a driver or a scoreboard. In Transaction-level modeling multiple processes communicate with each other by sending transactions through channels. For example, to transfer a transaction from generator to a driver, we don't need to send at signal level. To transfer a transaction from Driver to DUT, physical signals are used. The channel transfers transactions between the verification components, and it serves as a synchronizing agent between them.



Channels are similar to SystemVerilog mailboxes with advanced features. Vmm Channels provides much richer feature functionality than a SV mailbox. vmm channels are superset of mailboxes.

Some of the the benefits of channels over mailboxes:

- ⌚ Dynamic reconfiguration
- ⌚ Inbuilt notifications
- ⌚ Strict type checking
- ⌚ Out of order usage
- ⌚ task tee() for easy scoreboarding
- ⌚ Record and playback
- ⌚ task sneak() for monitors

Using `vmm\_channel() macro , channels can be created.

``vmm_channel(Custom_vmm_data)`

The above macro creates a channel Custom\_vmm\_data\_channel . There are various methods to access the channels.

In the following example, we will see

- 1) Channel creation using macros.
- 2) Constructing a channel.
- 3) Pushing a transaction in to channel.
- 4) Popping out a transaction from the channel.

We will create a channel for vmm\_data for this example. Users can create a channel for any transaction which is derived from the vmm\_data class. You can try this example by creating channel for Packet class which is discussed in previous section.

- 1) Define a channel using macro

```
`vmm_channel(vmm_data)
```

- 2) Construct an object of channel which is defined by the above macro.

```
vmm_data_channel p_c = new("p_c","chan",10);
```

- 3) Push a packet p\_put in to p\_c channel.

```
p_c.put(p_put);
```

- 4) Get a packet from p\_c channel.

```
p_c.get(p_get);
```

### Complete Example

```
`vmm_channel(vmm_data)
program test_channel();
  vmm_data p_put,p_get;
  vmm_data_channel p_c = new("p_c","chan",10);
  int i;
  initial
    repeat(10)
      begin
        #( $urandom()%10);
        p_put = new(null);
        p_put.stream_id = i++;
        $display(" Pushed a packet in to channel with id %d",p_put.stream_id);
        p_c.put(p_put); // Pushing a transaction in to channel
      end
end
```

```

initial
forever
  begin
    p_c.get(p_get); // popping a transaction from channel.
    $display(" Popped a packet from channel with id %d",p_get.stream_id);
  end
endprogram

```

[Download the file](#)

[vmm\\_channel.tar](#)

[Browse the code in vmm\\_channel.tar](#)

[Command to run the simulation](#)

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

[Log report](#)

Pushed a packet in to channel with id	0
Popped a packet from channel with id	0
Pushed a packet in to channel with id	1
Popped a packet from channel with id	1
Pushed a packet in to channel with id	2
Popped a packet from channel with id	2
Pushed a packet in to channel with id	3
Popped a packet from channel with id	3
Pushed a packet in to channel with id	4
Popped a packet from channel with id	4
Pushed a packet in to channel with id	5
Popped a packet from channel with id	5
Pushed a packet in to channel with id	6
Popped a packet from channel with id	6
Pushed a packet in to channel with id	7
Popped a packet from channel with id	7
Pushed a packet in to channel with id	8
Popped a packet from channel with id	8
Pushed a packet in to channel with id	9
Popped a packet from channel with id	9

[Vmm Channel Methods.](#)

```

function new ( string name, string instance,
int unsigned full = 1, int unsigned empty = 0, bit fill_as_bytes = 0 );
function void reconfigure (int full = -1, int empty = -1, logic fill_as_bytes = 1'bX );
function int unsigned full_level ( );
function int unsigned empty_level ( );
function int unsigned level ( );

```

```

function int unsigned size ( );
function bit is_full ( );
function void flush ( );
function void sink ( );
function void flow ( );
function void lock ( bit [1:0] who );
function void unlock ( bit [1:0] who );
function bit is_locked ( bit [1:0] who );
task put ( class_name obj, int offset = -1 );
function void sneak ( class_name obj, int offset = -1 );
function class_name unput ( int offset = -1 );
task get ( output class_name obj, input int offset = 0 );
task peek ( output class_name obj, input int offset = 0 );
task activate ( output class_name obj, input int offset = 0 );
function class_name active_slot ( );
function class_name start ( );
function class_name complete ( vmm_data status = null );
function class_name remove ( );
function active_status_e status ( );
task tee ( output class_name obj );
function bit tee_mode ( bit is_on );
function void connect ( vmm_channel downstream );
function class_name for_each ( bit reset = 0 );
function int unsigned for_each_offset ( );
function bit record ( string filename );
task bit playback ( output bit success, input string filename,
input vmm_data loader, input bit metered = 0 );

```

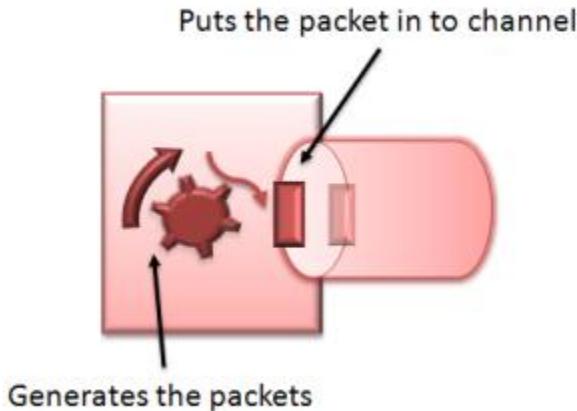
Refer to VMM book for details of each method.

### **VMM ATOMIC GENERATOR**

VMM has two types of generators. Atomic generator and Scenario generator.

Atomic generator is a simple generator which generates transactions randomly.

`vmm\_atomic\_gen is a macro which is used to define a class named <class\_name>\_atomic\_gen for any user-specified class derived from vmm\_data, using a process similar to the `vmm\_channel macro.



© www.testbench.in

To use `<class_name>_atomic_gen` class, a `<class_name>_channel` must exist. `<class_name>_atomic_gen` generates transactions and pushes it to `<class_name>_channel`. A `<class_name>_channel` object can be passed to generator while constructing.

```
function new(string instance,
           int stream_id = -1,
           <class_name>_channel out_chan = null);
```

The generator will stop generating the transaction after generating `stop_after_n_insts`. User can set the `stop_after_n_insts` to any unsigned int value. By default this values is 0.

- ☞ If `stop_after_n_insts` is 0, then generator generates infinite number of transactions.
- ☞ If `stop_after_n_insts` is non zero positive number, then generator generates `stop_after_n_insts` transactions.

We will see an example of creating a atomic generator for a packet class. Packet class is in file [Packet.sv](#).

1) define `vmm\_atomic\_gen macro for packet class. This macro creates a `packet_atomic_gen` class creates and randomizes packet transactions.

```
`vmm_atomic_gen(packet,"packet atomic generator")
```

2) define `vmm\_channel for the packet class. This macro creates a `packet_channel` which will be used by the `packet_atomic_gen` to store the transactions. Any other component can take the transactions from this channel.

```
`vmm_channel(packet)
```

3) Create an object of pcakt\_atomic\_gen.

```
packet_atomic_gen pkt_gen = new("Atomic Gen","test");
```

4) Set the number of transactions to be generated to 4

```
pkt_gen.stop_after_n_insts = 4;
```

5) Start the generator to generate transactions. These transactions are available to access through pkt\_chan as soon as they are generated.

```
pkt_gen.start_xactor();
```

6) Collect the packets from the pkt\_chan and display the packet content to terminal.

```
pkt_gen.out_chan.get(pkt);
```

```
pkt.display();
```

### Completed Example

```
`vmm_channel(Packet)
`vmm_atomic_gen(Packet,"Atomic Packet Generator")
program test_atomic_gen();
    Packet_atomic_gen pkt_gen = new("Atomic Gen","test");
    Packet pkt;
    initial
        begin
            pkt_gen.stop_after_n_insts = 4;
            #100; pkt_gen.start_xactor();
        end
    initial
        #200 forever
        begin
            pkt_gen.out_chan.get(pkt);
            pkt.display();
        end
    endprogram
```

### Download the example

[vmm\\_atomic\\_gen.tar](#)

[Browse the code in vmm\\_atomic\\_gen.tar](#)

### Commands to run the simulation

```
vcs -sverilog -f filelist -R -ntb_opts rvm -ntb_opts dtm
```

### Log file report

```
packet #1952805748.0.0
```

```
da:0xdb
```

```
sa:0x71
```

```
length:0x0e (data.size=1)
```

```
data[0]:0x63
```

```
fcs:0xe9
packet #1952805748.0.1
da:0xa7
sa:0x45
length:0xa4 (data.size=5)
data[0]:0x00 data[1]:0x4f .... data[3]:0xe7 data[4]:0xd8
fcs:0x31
```

```
packet #1952805748.0.2
da:0x15
sa:0xe6
length:0xa1 (data.size=1)
data[0]:0x80
fcs:0x01
```

```
packet #1952805748.0.3
da:0xd7
sa:0xa9
length:0xdc (data.size=3)
data[0]:0xcc data[1]:0x7c data[2]:0x7c
fcs:0x67
```

## **VMM\_XACTOR**

This base class is to be used as the basis for all transactors, including bus-functional models, monitors and generators. It provides a standard control mechanism expected to be found in all transactors.

**© virtual function void start\_xactor();**

Starts the execution threads in this transactor instance. This method is called by Environment class which is extended from vmm\_env start() method.

**© virtual function void stop\_xactor();**

Stops the execution threads in this transactor instance. This method is called by Environment class which is extended from vmm\_env stop() method.

**© protected task wait\_if\_stopped()**

protected task wait\_if\_stopped\_or\_empty(vmm\_channel chan)

Blocks the thread execution if the transactor has been stopped via the stop\_xactor() method or if the specified input channel is currently empty.

**© protected virtual task main();**

This task is forked off whenever the start\_xactor() method is called. It is terminated whenever the reset\_xactor() method is called. The functionality of a user-defined transactor must be implemented in this method.

Let us see an example of using vmm\_xactor.

1) Extend vmm\_xactor to create a custom\_xtor.

```
class Driver extends vmm_xactor;
```

2) Define constructor. In this example, we don't have any interfaces of channels, we will not implement them.

Call the super.new() method.

```
function new();
    super.new("Driver Transactor", "inst", 0);
endfunction: new
```

3) Define main() method.

First call the super.main() method.

Then define the activity which you want to do.

```
task main();
    super.main();
    forever begin
        #100;
        $display(" Driver : %d",$time);
    end
endtask: main
```

Now we will see how to use the above defined Custom\_xtor class.

1) Create a object of Custom\_xtor.

```
Driver drvr = new();
```

2) Call the start\_xactor() methods of Custom\_xtor object.

Now the main() method which is defined starts gets executed.

```
#100 drvr.start_xactor();
```

3) Call the stop\_xactor() methos.

This will stop the execution of main() method.

```
#1000 drvr.stop_xactor();
```

## Complete Vmm\_xactor Example

```
class Driver extends vmm_xactor;
function new();
    super.new("Driver Transactor", "inst", 0);
endfunction: new
task main();
    super.main();
```

```

forever begin
    #100;
    $display(" Driver : %d",$time);
end
endtask: main
endclass:Driver
program test();
    Driver drvr = new();
    initial
        fork
            #100 drvr.start_xactor();
            #1000 drvr.stop_xactor();
        join
endprogram

```

[Download the files](#)

[vmm\\_xactor.tar](#)

[Browse the code in vmm\\_xactor.tar](#)

[Command to run the simulation](#)

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

### [Log file report](#)

```

Driver :      200
Driver :      300
Driver :      400
Driver :      500
Driver :      600
Driver :      700
Driver :      800
Driver :      900
Driver :     1000
$finish at simulation time      1000

```

### [Vmm\\_xactor Members](#)

```

function new ( string name, string instance, int stream_id = -1 );
virtual function string get_name ( );
virtual function string get_instance ( );
vmm_log log;
int stream_id;
virtual function void prepend_callback ( vmm_xactor_callbacks cb );
virtual function void append_callback ( vmm_xactor_callbacks cb );
virtual function void unregister_callback ( vmm_xactor_callbacks cb );
vmm_notify notify;

```

```

// Enumeration values for the state of the transactor:
enum { XACTOR_IDLE, XACTOR_BUSY,
XACTOR_STARTED, XACTOR_STOPPED, XACTOR_RESET };

virtual function void start_xactor ();
virtual function void stop_xactor ();
virtual function void reset_xactor (reset_e rst_typ = SOFT_RST );
protected task wait_if_stopped ();
protected task wait_if_stopped_or_empty (vmm_channel chan );
protected virtual task main ();
virtual function void save_rng_state ();
virtual function void restore_rng_state ();
virtual function void xactor_status (string prefix = "");

// Macro to simplify the calling of callback methods:
`vmm_callback (callback_class_name, method (args ))

```

## **VMM CALLBACK**

Callback mechanism is used for altering the behavior of the transactor without modifying the transactor. One of the many promises of Object-Oriented programming is that it will allow for plug-and-play re-usable verification components. Verification Designers will hook the transactors together to make a verification environment. In SystemVerilog, this hooking together of transactors can be tricky. Callbacks provide a mechanism whereby independently developed objects may be connected together in simple steps.

This article describes vmm callbacks. Vmm callback might be used for simple notification, two-way communication, or to distribute work in a process. Some requirements are often unpredictable when the transactor is first written. So a transactor should provide some kind of hooks for executing the code which is defined afterwards. In vmm, these hooks are created using callback methods. For instance, a driver is developed and an empty method is called before driving the transaction to the DUT. Initially this empty method does nothing. As the implementation goes, user may realize that he needs to print the state of the transaction or to delay the transaction driving to DUT or inject an error into transaction. Callback mechanism allows executing the user defined code in place of the empty callback method. Other example of callback usage is in monitor. Callbacks can be used in a monitor for collecting coverage information or for hooking up to scoreboard to pass transactions for self checking. With this, user is able to control the behavior of the transactor in verification environment and individual testcases without doing any modifications to the transactor itself.

Following are the steps to be followed to create a transactor with callbacks. We will see simple example of creating a Driver transactor to support callback mechanism.

- 1) Define a facade class.

Extend the vmm\_xactor\_callbacks class to create a faced class.

Define required callback methods. All the callback methods must be virtual.

In this example, we will create callback methods which will be called before driving the packet and after driving the packet to DUT.

```
class Driver_callbacks extends vmm_xactor_callbacks;  
  
virtual task pre_send(); endtask  
virtual task post_send(); endtask
```

**endclass**

2) Calling callback method.

Inside the transactor, callback methods should be called whenever something interesting happens.

We will call the callback method before driving the packet and after driving the packet. We defined 2 methods in facade class. We will call pre\_send() method before sending the packet and post\_send() method after sending the packet.

Using a `vmm\_callback(,) macro, callback methods are called.

There are 2 arguments to `vmm\_callback(,) macro.

First argument must be the facade class.

Second argument must be the callback method in the facade class.

To call pre\_send() method , use macro

`vmm\_callback(Driver\_callbacks,pre\_send());

and similarly to call post\_send() method,

`vmm\_callback(Driver\_callbacks,post\_send());

Place the above macros before and after driving the packet.

```
virtual task main();
```

```
super.main();
```

```
forever begin
```

```
`vmm_callback(Driver_callbacks,pre_send());
```

```
$display(" Driver: Started Driving the packet ..... %d",$time);
```

```
// Logic to drive the packet goes here
```

```
// let's consider that it takes 40 time units to drive a packet.
```

```
#40;
```

```
$display(" Driver: Finished Driving the packet ..... %d",$time);
```

```
`vmm_callback(Driver_callbacks,post_send());
```

```
end
endtask
```

With this, the Driver implementation is completed with callback support.

### Complete Source Code

```
class Driver_callbacks extends vmm_xactor_callbacks;

virtual task pre_send(); endtask
virtual task post_send(); endtask
endclass

class Driver extends vmm_xactor;
function new();
super.new("Driver","class");
endfunction
virtual task main();
super.main();
forever begin
`vmm_callback(Driver_callbacks,pre_send());
$display(" Driver: Started Driving the packet ..... %d",$time);
// Logic to drive the packet goes here
// let's consider that it takes 40 time units to drive a packet.
#40;
$display(" Driver: Finished Driving the packet ..... %d",$time);
`vmm_callback(Driver_callbacks,post_send());
end
endtask
endclass
```

Let's run the driver in simple testcase. In this testcase, we are not changing any callback methods definitions.

### Testcase 1 Source Code

```
program testing_callbacks();
Driver drvr = new();
initial
begin
#100 drvr.start_xactor();
#200 drvr.stop_xactor();
end
endprogram
```

## Download files

[vmm\\_callback.tar](#)

[Browse the code in vmm\\_callback.tar](#)

## Command to run the simulation

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

## Log report

Driver: Started Driving the packet .....	100
Driver: Finished Driving the packet .....	140
Driver: Started Driving the packet .....	140
Driver: Finished Driving the packet .....	180
Driver: Started Driving the packet .....	180
Driver: Finished Driving the packet .....	220
Driver: Started Driving the packet .....	220
Driver: Finished Driving the packet .....	260
Driver: Started Driving the packet .....	260
Driver: Finished Driving the packet .....	300
Driver: Started Driving the packet .....	300
\$finish at simulation time	300

Following steps are to be performed for using callback mechanism to do required functionality.

We will see how to use the callbacks which are implemented in the above defined driver in a testcase.

1) Implement the user defined callback method by extending facade class of the driver class.

We will delay the driving of packet be 20 time units using the pre\_send() call back method.

We will just print a message from post\_send() callback method.

```
class Custom_Driver_callbacks_1 extends Driver_callbacks;
virtual task pre_send();
$display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
#20;
endtask
virtual task post_send();
$display("CB_1:post_send: Just a message from post send callback method \n");
endtask
```

2) Construct the user defined facade class object.

```
Custom_Driver_callbacks CDc = new();
```

3) Register the callback method. vmm\_xactor class has append\_callback() which is used to register the callback.

```
drvr.append_callback(CDc_1);
```

## Testcase 2 Source Code

```

program testing_callbacks();
  Driver drvr = new();
  class Custom_Driver_callbacks_1 extends Driver_callbacks;
    virtual task pre_send();
      $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
      #20;
    endtask
    virtual task post_send();
      $display("CB_1:post_send: Just a message from post send callback method \n");
    endtask
  endclass
Custom_Driver_callbacks_1 CDc_1 = new();
initial
  begin
    drvr.append_callback(CDc_1);
    #100 drvr.start_xactor();
    #200 drvr.stop_xactor();
  end
endprogram

```

#### Download the example

[vmm\\_callback\\_1.tar](#)

[Browse the code in vmm\\_callback\\_1.tar](#)

#### Simulation Command

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

Run the testcase. See the log results; We delayed the driving of packet by 20 time units using callback mechanism. See the difference between the previous testcase log and this log.

#### Log report

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 100

Driver: Started Driving the packet ..... 120

Driver: Finished Driving the packet ..... 160

CB\_1:post\_send: Just a message from post send callback method

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 160

Driver: Started Driving the packet ..... 180

Driver: Finished Driving the packet ..... 220

CB\_1:post\_send: Just a message from post send callback method

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 220

Driver: Started Driving the packet ..... 240

Driver: Finished Driving the packet ..... 280  
CB\_1:post\_send: Just a message from post send callback method

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 280  
Driver: Started Driving the packet ..... 300  
\$finish at simulation time 300

Now we will see registering 2 callback methods.

1) Define another user defined callback methods by extending facade class.

```
class Custom_Driver_callbacks_2 extends Driver_callbacks;  
  virtual task pre_send();  
    $display("CB_2:pre_send: Hai .... this is from Second callback %d",$time);  
  endtask  
endclass
```

2) Construct the user defined facade class object.

```
Custom_Driver_callbacks_2 CDc_2 = new();
```

3) Register the object

```
drvr.append_callback(CDc_2);
```

### Testcase 3 Source Code

```
program testing_callbacks();  
  Driver drvr = new();  
  class Custom_Driver_callbacks_1 extends Driver_callbacks;  
    virtual task pre_send();  
      $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);  
      #20;  
    endtask  
  
    virtual task post_send();  
      $display("CB_1:post_send: Just a message from post send callback method \n");  
    endtask  
  
  endclass  
  
  class Custom_Driver_callbacks_2 extends Driver_callbacks;  
    virtual task pre_send();  
      $display("CB_2:pre_send: Hai .... this is from Second callback %d",$time);  
    endtask  
  
  endclass
```

```

Custom_Driver_callbacks_1 CDc_1 = new();
Custom_Driver_callbacks_2 CDc_2 = new();

initial
begin
    drvr.append_callback(CDc_1);
    drvr.append_callback(CDc_2);
    #100 drvr.start_xactor();
    #200 drvr.stop_xactor();
end
endprogram

```

### Download source code

[vmm\\_callback\\_2.tar](#)

[Browse the code in vmm\\_callback\\_2.tar](#)

### Command to run the simulation

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

Run the testcase and analyze the result.

### Log report

CB_1:pre_send: Delaying the packet driving by 20 time units.	100
CB_2:pre_send: Hai .... this is from Second callback	120
Driver: Started Driving the packet .....	120
Driver: Finished Driving the packet .....	160
CB_1:post_send: Just a message from post send callback method	
CB_1:pre_send: Delaying the packet driving by 20 time units.	160
CB_2:pre_send: Hai .... this is from Second callback	180
Driver: Started Driving the packet .....	180
Driver: Finished Driving the packet .....	220
CB_1:post_send: Just a message from post send callback method	
CB_1:pre_send: Delaying the packet driving by 20 time units.	220
CB_2:pre_send: Hai .... this is from Second callback	240
Driver: Started Driving the packet .....	240
Driver: Finished Driving the packet .....	280
CB_1:post_send: Just a message from post send callback method	
CB_1:pre_send: Delaying the packet driving by 20 time units.	280

```

CB_2:pre_send: Hai .... this is from Second callback      300
Driver: Started Driving the packet .....      300
$finish at simulation time      300

```

The log results show that pre\_send() method of CDc\_1 is called first and then pre\_send() method of Cdc\_2. This is because of the order of the registering callbacks.

```

drvr.append_callback(CDc_1);
drvr.append_callback(CDc_2);

```

Now we will see how to change the order of the callback method calls.

Use prepend\_callback() method for registering instead of append\_callback() method.

#### Testcase 4 Source Code

```

program testing_callbacks();
  Driver drvr = new();
  class Custom_Driver_callbacks_1 extends Driver_callbacks;
    virtual task pre_send();
      $display("CB_1:pre_send: Delaying the packet driving by 20 time units. %d",$time);
      #20;
    endtask
    virtual task post_send();
      $display("CB_1:post_send: Just a message from post send callback method \n");
    endtask
  endclass
  class Custom_Driver_callbacks_2 extends Driver_callbacks;
    virtual task pre_send();
      $display("CB_2:pre_send: Hai .... this is from Second callback %d",$time);
    endtask
  endclass
  Custom_Driver_callbacks_1 CDc_1 = new();
  Custom_Driver_callbacks_2 CDc_2 = new();
  initial
    begin
      drvr.append_callback(CDc_1);
      drvr.prepend_callback(CDc_2);
      #100 drvr.start_xactor();
      #200 drvr.stop_xactor();
    end
  endprogram

```

## Download the source code

[vmm\\_callback\\_3.tar](#)

[Browse the code in vmm\\_callback\\_3.tar](#)

## Command to run the simulation

vcs -sverilog -f filelist -R -ntb\_opts rvm -ntb\_opts dtm

Run and analyze the results.

Log results show that, pre\_send() method of CDs\_1 is called after calling CDs\_2 pre\_send() method.

## Log file report

CB\_2:pre\_send: Hai .... this is from Second callback 100

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 100

Driver: Started Driving the packet ..... 120

Driver: Finished Driving the packet ..... 160

CB\_1:post\_send: Just a message from post send callback method

CB\_2:pre\_send: Hai .... this is from Second callback 160

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 160

Driver: Started Driving the packet ..... 180

Driver: Finished Driving the packet ..... 220

CB\_1:post\_send: Just a message from post send callback method

CB\_2:pre\_send: Hai .... this is from Second callback 220

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 220

Driver: Started Driving the packet ..... 240

Driver: Finished Driving the packet ..... 280

CB\_1:post\_send: Just a message from post send callback method

CB\_2:pre\_send: Hai .... this is from Second callback 280

CB\_1:pre\_send: Delaying the packet driving by 20 time units. 280

Driver: Started Driving the packet ..... 300

\$finish at simulation time 300

Vmm also provides method to remove the callback methods which are registered.

## **VMM TEST**

vmm\_test is introduced in vmm 1.1.

To know the vmm version which you are using, use this command

```
vcs -R -sverilog -ntb_opts dtm
```

```
+incdir+$VMM_HOME/sv $VMM_HOME/sv/vmm_versions.sv
```

vmm\_test is used for compiling all the testcases in one compilation. The simulation of each testcase is done individually. Traditionally for each testcase, compile and simulation are done per testcase file. With this new approach, which dose compilation only once, will save lot of cup.

Generally each testcase can be divided into two parts.

### **Procedural code part.**

The procedural code part (like passing the above new constrained transaction definition to some atomic generator, calling the env methods etc) has to be defined between these macros. vmm provides 2 macros to define testcase procedural part.

```
`vmm_test_begin(testcase_name,vmm_env,"Test Case Name String")  
`vmm_test_env(testcase_name)
```

### **Declarative code part.**

The declarative part( like new constrained transacting definition) is defined outside these macros.

## **Writing A Testcase**

Let us see an example of writing a testcase.

Inside the testcase, we will define a new transaction and pass this transaction to the atomic generator.

### **Declarative part:**

1) Define all the declarative code.

```
class constrained_tran extends pcie_transaction;  
  // Add some constraints  
  // Change some method definitions  
end class
```

2) Define a handle to the above object.

```
constrained_tran c_tran_obj;
```

### **Procedural part:**

Use a `vmm\_test\_begin . There are 3 arguments to macro.

①

The first argument is the name of the testcase class and will also be used as the name of the testcase in the global testcase registry.

②

The second argument is the name of the environment class that will be used to execute the testcase. A data member of that type named "env" will be defined and assigned, ready to be

used.



The third argument is a string documenting the purpose of the test.

```
`vmm_test_begin(test_1,vmm_env,"Test_1")
```

In this testcase, we want to pass the c\_tran\_obj object. The steps t pass the c\_tran\_obj as per the vmm\_env is

```
env.build();  
c_tran_obj = new(" ");  
env.atomic_gen.randomized_obj = c_tran_obj;
```

Start the vmm\_env execution.

```
env.run();
```

Now use `vmm\_test\_end to define the end of the testcase. The argument to this is the testcase name.

```
`vmm_test_end(test_1)
```

Following is the full testcase source code which we discussed above.

#### Testcase source code

```
class constrained_tran extends pcie_transaction;  
end class  
constrained_tran c_tran_obj  
`vmm_test_begin(test_1,vmm_env,"Test_1")  
$display(" Start of Testcase : Test_1 ");  
env.build();  
c_tran_obj = new(" ");  
env.atomic_gen.randomized_obj = c_tran_obj;  
env.run();  
$display(" End of Testcase : Test_1 ");  
`vmm_test_end(test_1)
```

Like this you can write many testcases in separate file or single file.

#### Example Of Using Vmm test

Now we will implement 3 simple testcases and a main testcase which will be used for selecting any one of the 3 testcases.

#### testcase 1

Write this testcase in a testcase\_1.sv file

```
// Define all the declarative code hear. I done have anything to show you.  
//  
// class constrained_tran extends pcie_transaction;  
//  
// end class  
//
```

```

// constrained_tran c_tran_obj
`vmm_test_begin(test_1,vmm_env,"Test_1")
$display(" Start of Testcase : Test_1 ");
// This is procedural part. You can call build method.
env.build();
// You can pass the above created transaction to atomic gen.
// c_tran_obj = new(" ");
// env.atomic_gen.randomized_obj = c_tran_obj;
//
//
env.run();
$display(" End of Testcase : Test_1 ");
`vmm_test_end(test_1)

```

### testcase 2

Write this testcase in a testcase\_2.sv file

```

`vmm_test_begin(test_2,vmm_env,"Test_2")
$display(" Start of Testcase : Test_2 ");
// Do something like this ....
env.build();
// like this ....
env.run();
$display(" End of Testcase : Test_2 ");
`vmm_test_end(test_2)

```

### testcase 3

Write this testcase in a testcase\_3.sv file

```

`vmm_test_begin(test_3,vmm_env,"Test_3")
$display(" Start of Testcase : Test_3 ");
// Do something like this ....
env.build();
// like this ....
env.run();
$display(" End of Testcase : Test_3 ");
`vmm_test_end(test_3)

```

### main testcase

Now we will write the main testcase. This doesn't have any test scenario, it is only used for handling the above 3 testcases.

This should be written in program block.

- 1) First include all the above testcases inside the program block.

```

`include "testcase_1.sv"
`include "testcase_2.sv"

```

```
`include " testcase_3.sv"
```

2) Define env class object.

```
vmm_env env = new();
```

As I dont have a custom env class to show, I used vmm\_env. You can use your custom defined env.

3) In the initial block call the run() method of vmm\_test\_registry class and pass the above env object as argument. This run method of vmm\_test\_registry is a static method, so there is no need to create an object of this class.

```
vmm_test_registry::run(env);
```

### Main testcase source code

```
`include "vmm.sv"  
program main();  
`include " testcase_1.sv"  
`include " testcase_2.sv"  
`include " testcase_3.sv"  
vmm_env env;  
initial  
begin  
$display(" START OF TEST CASE ");  
env = new();  
vmm_test_registry::run(env);  
$display(" START OF TEST CASE ");  
end  
endprogram
```

Now compile the above code using command.

```
vcs -sverilog main_testcase.sv +incdir+$VMM_HOME/sv
```

Now simulate the above compiled code.

To simulate , just do ./simv and see the log.

You can see the list of the testcases. This simulation will not eecute any testcase.

### LOG

START OF TEST CASE

```
*FATAL*[FAILURE] on vmm_test_registry(class) at 0:
```

No test was selected at runtime using +vmm\_test=<test>.

Available tests are:

- 0) Default : Default testcase that simply calls env::run()
- 1) test\_1 : Test\_1
- 2) test\_2 : Test\_2
- 3) test\_3 : Test\_3

To run testcase\_1 use ./simv +vmm\_test=test\_1

### LOG

START OF TEST CASE

Normal[NOTE] on vmm\_test\_registry(class) at 0:  
Running test "test\_1"..."  
Start of Testcase : Test\_1  
Simulation PASSED on ./ (./) at 0 (0 warnings, 0 demoted errors & 0 demoted warnings)

End of Testcase : Test\_1

START OF TEST CASE

Simillarly to run testcase\_2 ./simv +vmm\_test=test\_2

### LOG

START OF TEST CASE

Normal[NOTE] on vmm\_test\_registry(class) at 0:  
Running test "test\_2"..."  
Start of Testcase : Test\_2  
Simulation PASSED on ./ (./) at 0 (0 warnings, 0 demoted errors & 0 demoted warnings)

End of Testcase : Test\_2

START OF TEST CASE

You can also call the Default testcase, which just calls the env::run()  
./simv +vmm\_test=Default

### LOG

START OF TEST CASE

Normal[NOTE] on vmm\_test\_registry(class) at 0:  
Running test "Default"..."  
Simulation PASSED on ./ (./) at 0 (0 warnings, 0 demoted errors & 0 demoted warnings)

START OF TEST CASE

Download the source code

vmm\_test.tar

Browse the code in vmm\_test.tar

Command to compile

vcs -sverilog -ntb\_opts dtm +incdir+\$VMM\_HOME/sv main\_testcase.sv

Commands to simulate testcases

./simv

./simv +vmm\_test=Default

./simv +vmm\_test=test\_1

./simv +vmm\_test=test\_2

./simv +vmm\_test=test\_3

## **VMM CHANNEL RECORD AND PLAYBACK**

The purpose of this channel feature is to save all the incoming transaction to the specified file and play back the transaction from the file.

This will be useful when we want to recreate a random scenario in different environment. For example, you found a very interesting random scenario in 10G speed Ethernet. Now you want to use the same random scenario on 1G Ethernet. It may not be possible to reproduce the same scenario at 1G speed even with the SV random stability. The other way is to write a directed testcase to recreate the same scenario. Using this Record/playback feature, we can record the transaction in the 10G speed Ethernet and then play this scenario at 1G speed Ethernet.

One more advantage of this feature is, it allows us to shutdown the activities of some components. For Example, consider a layered protocol, the Ethernet. The Verification environment can be developed with layers of transactors corresponding to each protocol layer like, TCP over IP layer, IP fragment and reassembly layer, IP fragment over Ethernet and Ethernet layer. All these layers will be connected using channels. With record/playback feature, the input transaction to Ethernet layer can be recorded and in the next simulation, we can shutdown the TCP over IP layer, IP fragment and reassembly layer, IP fragment over Ethernet layers transactors and by playing back the recorded transactions. This will improve the memory consumption and execution speed.

Record/Playback is independent of Random stability. Changing the seed value will not affect the playback. This will give us an opportunity to test the DUT using a specific scenario of the high layer (Consider the previous example: Some interested scenario in IP fragment over Ethernet) with lot of different randomization using different seeds of lower layer protocol (Ethernet layer). This will also help, when we cannot reproduce the random scenario using Seed. If you found a Bug in the DUT, and still you are in the development stage of your environment, you cannot stop the development in the environment till the bug is fixed. There are chances that the changes in the environment will not allow you to reproduce the same scenario even with the same seed. In this case, if you have stored the transactions, you can playback the transaction.

Now we will learn about the implementation.

### **Recording**

To record the flow of transaction descriptors added through the channel instance, call the record method of that instance.

**function bit record(string filename);**

The vmm\_data::save() method must be implemented for recording. All the transaction which are added to channel using the vmm\_channel::put() method are recorded and are available for playback. A null filename stops the recording process. vmm\_channel::record() method returns TRUE if the specified file was successfully opened.

RECORDING notification is indicated by vmm\_channel::notify, when recording.

### **Playing Back**

To playback the transactions in the recorded file, use the vmm\_channel::playback() method.

```

task playback(output bit success,
    input string filename,
    input vmm_data factory,
    input bit metered = 0,
    input vmm_scenario grabber = null);

```

This method will block the sources of the current channel like vmm\_channel::put() method till the playback is completed. The vmm\_data::load() method must be implemented for playback. The order of the transaction will be same as the order while recording. If the metered argument is TRUE, the transaction descriptors are added to the channel with the same relative time interval as they were originally put in when the file was recorded.

The success argument is set to TRUE if the playback was successful. If the playback process encounters an error condition such as a NULL (empty string) filename, a corrupt file or an empty file, then success is set to FALSE.

PLAYBACK\_DONE notification is indicated by vmm\_channel::notify, when playback is completed.

### EXAMPLE

Let us write an example to see how the record and playback works.

Record & Playback of channels can be used with atomic generator and scenario generators. Here is a simple transaction with its channel and atomic generator. We will use this atomic generator to generate transaction. The atomic generator has out\_chan channel, this channel is used send the generated transaction. We will record and playback the transaction in the out\_chan. I used Vmm\_data macros for implementing all the required methods.

```

class trans extends vmm_data;
typedef enum bit {BAD_FCS=1'b0, GOOD_FCS=1'b1} kind_e;
rand bit [47:0] DA;
rand bit [47:0] SA;
rand int length;
rand kind_e kind;
`vmm_data_member_begin(trans)
    `vmm_data_member_scalar(DA, DO_ALL)
    `vmm_data_member_scalar(SA, DO_ALL)
    `vmm_data_member_scalar(length, DO_ALL)
    `vmm_data_member_enum(kind, DO_ALL)
`vmm_data_member_end(trans)
endclass
// for declaring trans_channel
`vmm_channel(trans)

```

```
// for declaring trans_atomic_gen
`vmm_atomic_gen(trans, "Gen")
```

Now we will write a simple program to record and play back the above defined transaction. Use the trans\_atomic\_gen to generate the transactions. This also puts the transactions into out\_chan channel. Call the start\_xactor() method of the generator to start the generator of transaction. Call the record() method of out\_chan to record the incoming transactions.

```
gen.stop_after_n_insts = 3;
    gen.start_xactor();
    gen.out_chan.record("Trans_recordes.tr");
```

To play back the recorded transactions, call the playback method. Then check if the playback is fine.

```
tr = new();
    gen.out_chan.playback(success, "Trans_recordes.tr", tr);
    if(!success)
        $display(" Failed to playback Trans_recordes.tr"); ]
```

Now we will write a logic which gets the transaction from the channel and prints the content to terminal, so that we can observe the results.

The above logic generates 3 transactions. Get the 3 transaction from the channel and print the content.

```
initial
repeat(3)
begin
#10;
gen.out_chan.get(tr);
tr.display();
end
```

#### Complete source code

```
program main();
trans tr;
trans_atomic_gen gen = new("Atomic Gen",1);
bit success;
// This is the producer logic.
// Transactions are produced from Atomic generator in RECORD mode.
// Transactions are produced from recorded file in PLAYBACK mode.

initial
begin
if($test$plusargs("RECORD"))
begin
$display(" RECORDING started ");
gen.stop_after_n_insts = 3;
```

```

gen.start_xactor();
gen.out_chan.record("Trans_recordes.tr");
end
else if($test$plusargs("PLAYBACK"))
begin
    $display(" PLAYBACK started ");
    tr = new();
    gen.out_chan.playback(success,"Trans_recordes.tr",tr);
    if(!success)
        $display(" Failed to playback Trans_recordes.tr");
end
else
    $display(" give +RECORD or +PLAYBACK with simulation command");
end
// This is consumer logic.
initial
repeat(3)
begin
    #10;
    gen.out_chan.get(tr);
    tr.display();
end
endprogram
Download the source code

```

This works with vmm 1.1 and above.

[vmm\\_record\\_playback.tar](#)

[Browse the code in vmm\\_record\\_playback.tar](#)

### Command to record the transaction

vcs -R -sverilog +includir+\$VMM\_HOME/sv record\_playback.sv +RECORD

### Log file

RECORDING started

trans (1.0.0):

```

DA='haecd55f5b651
SA='h13db0b1a590e
length='hd8a7d371
kind=GOOD_FCS

```

```
trans (1.0.1):  
DA='h5e980e9b7ce9  
SA='h44a4bbdaf703  
length='h5714c3a7  
kind=GOOD_FCS
```

```
trans (1.0.2):  
DA='hba4f5a021300  
SA='h2de750b6cc3e  
length='h22ca2bd8  
kind=GOOD_FCS
```

You can also see a file named "Trans\_recordes.tr" in your simulation directory.

### Command to playback the transaction

```
vcs -R -sverilog +inmdir+$VMM_HOME/sv trans.sv +PLAYBACK
```

### Log File

PLAYBACK started

```
trans (0.0.0):  
DA='haecd55f5b651  
SA='h13db0b1a590e  
length='hd8a7d371  
kind=GOOD_FCS
```

```
trans (0.0.0):  
DA='h5e980e9b7ce9  
SA='h44a4bbdaf703  
length='h5714c3a7  
kind=GOOD_FCS
```

```
trans (0.0.0):  
DA='hba4f5a021300  
SA='h2de750b6cc3e  
length='h22ca2bd8  
kind=GOOD_FCS
```

Look, the same transactions which were generated while RECORDING are played back.

## **VMM SCENARIO GENERATOR**

Atomic generator generates individual data items or transaction descriptors. Each item is generated independently of other items in a random fashion. Atomic generator is simple to describe and use.

Unlike atomic generator, a scenario generator generates a sequence of transaction.

### **Example of sequence of transaction:**

- ⌚ CPU instructions like LOAD\_A,LOAD\_B,ADD\_A\_B,STORE\_C.
- ⌚ Packets with incrementing length.
- ⌚ Do write operation to address "123" and then do the read to address "123".

It is very unlikely that atomic generator to generate transaction in the above ordered sequence.

With scenario generator, we can generate these sequence of transaction.

VMM provides `vmm\_scenario\_gen() macro for quickly creating a scenario generator.

`vmm\_scenario\_gen(class\_name, "Class Description");

The macro defines classes named <class\_name>\_scenario\_gen, <class\_name>\_scenario, <class\_name>\_scenario\_election , <class\_name>\_scenario\_gen\_callbacks and <class\_name>\_atomic\_scenario.

```
class custom_scenario_gen extends vmm_xactor;  
class custom_scenario extends vmm_data;  
class custom_atomic_scenario extends custom_scenario;  
class custom_scenario_election;  
class custom_scenario_gen_callbacks extends vmm_xactor_callbacks
```

### **<class name> scenario:**

This scenario generator can generate more than one scenario. Each scenario which can contain more than one transaction is described in a class which is extended from <class\_name>\_scenario.

For each scenario, following variables must be constraint.

- ⌚ Length: number of transactions in an array.
- ⌚ Repeated: number of times to repeat this scenario.

### **<class name> atomic scenario:**

This class is a atomic scenario. This is the default scenario. This scenario is a random transactions.

### **<class name> scenario election:**

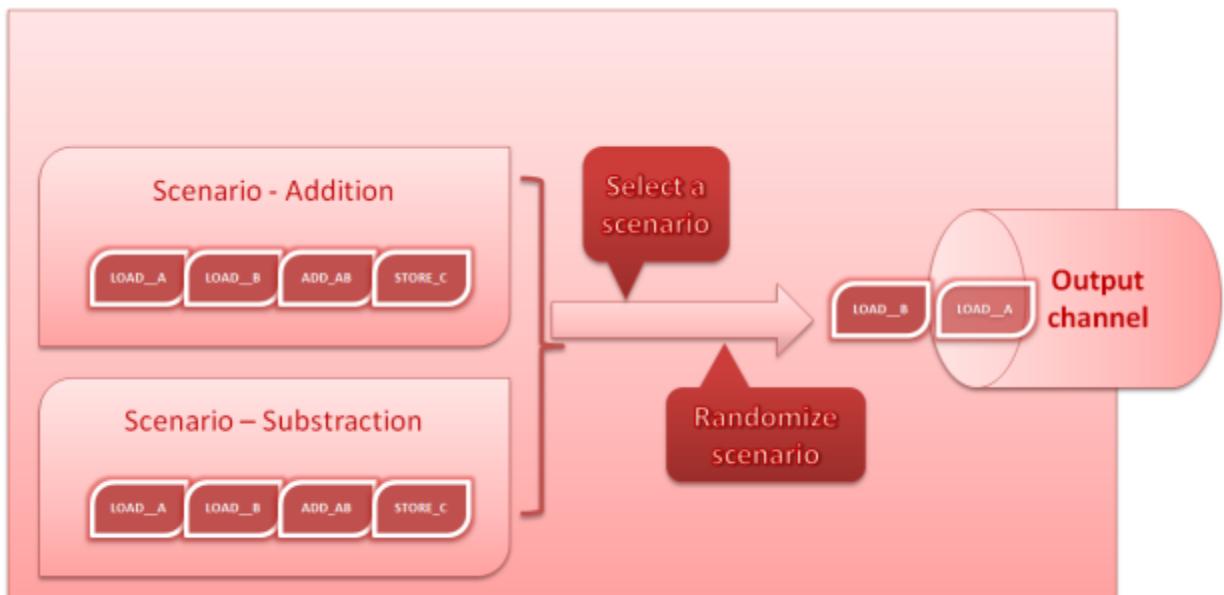
This class is the arbiter which determines the order that the known scenarios are applied. By default, scenarios are elected atomically. User can extend this class to define an order in which the scenarios should be picked.

### <class name> scenario\_gen:

This class is the scenario generator which generates the transactions and sends out using output channel. This class has a queue of scenario objects. Each scenario contains transaction instances in an array.

### <class name> scenario\_gen\_callbacks:

This class provides callback mechanism. There are two callback methods define. pre\_scenario\_randomize() and post\_scenario\_gen() which are called at pre-randomization of the scenario and post-generation of the scenario respectively.



© [www.testbench.in](http://www.testbench.in)

### Example

Let us write an example.

Following is the transaction class which we will use to write a scenario generator.

```
class instruction extends vmm_data;
    vmm_log log;
    typedef enum {LOAD_A,LOAD_B,ADD_A_B,SUB_A_B,STORE_C } kinds_e;
    rand kinds_e inst;
    function new();
        super.new(this.log);
    endfunction:new
    virtual function string psdisplay(string prefix = "");
        psdisplay = $psprintf(" Instruction : %s | stream_id : %0d | scenario_id : %0d
",inst.name(),stream_id,scenario_id);
    endfunction:psdisplay
```

```

virtual function vmm_data allocate();
    instruction tr = new;
    allocate = tr;
endfunction:allocate

virtual function vmm_data copy(vmm_data cpy = null);
    instruction to;
    if (cpy == null)
        to = new;
    else
        if (!$cast(to, cpy)) begin
            `vmm_fatal(this.log, "Attempting to copy to a non instruction instance");
            return null;
        end
    super.copy_data(to);
    to.inst = this.inst;
    copy = to;
endfunction:copy

```

**endclass**

The above transaction creates CPU instructions.

Let us consider that sequence of instruction LOAD\_A,LOAD\_B,ADD\_A\_B,STORE\_C is an interesting scenario and LOAD\_A,LOAD\_B,SUB\_A\_B,STORE\_C is also an interesting scenario.

When instructions are generated, we want to generate these 2 sequences of instruction. Let us see how to generate these 2 sequence of instructions .

As we have already discussed, `vmm\_scenario\_gen() creates use full class for scenario generation.

1) Use `vmm\_scenario\_gen() macro to declare the scenario classes.

`vmm\_scenario\_gen(instruction, "Instruction Scenario Generator")

This macro will create following classes

- ⌚ instruction\_scenario\_gen
- ⌚ instruction\_scenario
- ⌚ instruction\_atomic\_scenario
- ⌚ instruction\_scenario\_election
- ⌚ instruction\_scenario\_gen\_callbacks

2) Define interesting scenarios by extending inst\_scenario;

**class** instruction\_scenario\_add\_sub **extends** instruction\_scenario;

3) Define the first scenario. It is sequence of instructions for addition operation.

a) Declare a variable for identifying the scenario.

**int** addition\_scenario\_id ;

Each scenario has more than one inductions. All these instructions are in a queue "items" which is already defined in "instruction\_scenario".

The rand varible "scenario\_kind", which pre defined in the vmm\_scenario class, will randomly select one of the defined scenarios. "scenario\_kind" variable has the id of the current scenario. So we have to define the addition scenario, when the "scenario\_kind" value is addition\_scenario\_id.

b) Constrain the scenario kind,

**constraint** addition\_scenario\_items {

**if**(\$void(scenario\_kind) == addition\_scenario\_id) {

c) The number of instructions define in a scenario is specified by the predefined variable "length". In our example, we have 4 instructions. So constrain the length to 4.

    length == 4;

d) The predefined variable "repeated" used to control if the VMM scenario generator would run the scenario more than once each time it is created. We are not interested in repeating so, constrain it to 0

    repeated == 0;

e) Constrain the individual items based on the requirements if this scenario is selected. Our requirement in this example is that "inst" should follow the sequence

LOAD\_A,LOAD\_B,ADD\_A\_B,STORE\_C

**foreach**(items[i])

**if**(i == 0)

**this**.items[i].inst == instruction::LOAD\_A;

**else if**(i == 1)

**this**.items[i].inst == instruction::LOAD\_B;

**else if**(i == 2)

**this**.items[i].inst == instruction::ADD\_A\_B;

**else if**(i == 3)

**this**.items[i].inst == instruction::STORE\_C;

4)Define second scenario. It is Sequence of instructions for subtraction operation.

a) Declare a variable for identifying the scenario.

**int** subtraction\_scenario\_id ;

b) Constrain the scenario kind,

**constraint** subtraction\_scenario\_items {

**if**(\$void(scenario\_kind) == subtraction\_scenario\_id) {

c) Constrain the length

    length == 4;

d) Constrain the repeated

```
repeated == 0;
```

e) Constrain the items

```
foreach(items[i])
    if(i == 0)
        this.items[i].inst == instruction::LOAD_A;
    else if(i == 1)
        this.items[i].inst == instruction::LOAD_B;
    else if(i == 2)
        this.items[i].inst == instruction::SUB_A_B;
    else if(i == 3)
        this.items[i].inst == instruction::STORE_C;
```

5) Define constructor method.

a) call the super.new() method.

Get a unique Ids from the define\_scenario() method for each scenario.

define\_scenario() is predefined in \*\_scenario class.

```
function new();
    this.addition_scenario_id = define_scenario("ADDITION",4);
    this.subtraction_scenario_id = define_scenario("SUBTRACTION",4);
endfunction
```

With this, we completed the implementation of scenarios.

### Scenario Code

```
`vmm_scenario_gen(instruction,"Instruction Scenario Generator")
```

```
class instruction_scenario_add_sub extends instruction_scenario;
```

```
////////// ADDITION SCENARIO //////////
```

```
int addition_scenario_id ;
constraint addition_scenario_items {
    if($void(scenario_kind) == addition_scenario_id) {
        repeated == 0;
        length == 4;
        foreach(items[i])
            if(i == 0)
                this.items[i].inst == instruction::LOAD_A;
            else if(i == 1)
                this.items[i].inst == instruction::LOAD_B;
```

**endclass**

## Testcase

- Now we will write a test case to see how to above

  - 1) Declare scenario generator  
`instruction_scenario_gen gen;`
  - 2) Declare the scenario which we defined earlier.  
`instruction_scenario_add_sub sce_add_sub;`
  - 3) Construct the generator and scenarios.  
`gen = new("gen",0);  
sce add sub = new();`

4) set the number of instances and scenarios generated by generator to 20 and 4 respectevy.

```
gen.stop_after_n_insts = 20;  
gen.stop_after_n_scenarios = 4;
```

5) Scenario generators store all the scenarios in scenario\_set queue. So, we have to add the scenario which we constructed above to the queue.

```
gen.scenario_set[0] = sce_add_sub;
```

6) Start the generator

```
gen.start_xactor();
```

7) Similar t the Atomic generator, the transactions created by the scenario generator are sent to out\_chan channel.

Get the instructions from the out\_chan channel and print the content.

```
repeat(20) begin
```

```
    gen.out_chan.get(inst);
```

```
    inst.display();
```

#### Testcase code

```
program test();
```

```
instruction_scenario_gen gen;
```

```
instruction_scenario_add_sub sce_add_sub;
```

```
instruction inst;
```

```
initial
```

```
begin
```

```
    gen = new("gen",0);
```

```
    sce_add_sub = new();
```

```
//gen.log.set_verbosity(vmm_log::DEBUG_SEV,"./","./");
```

```
    gen.stop_after_n_insts = 20;
```

```
    gen.stop_after_n_scenarios = 4;
```

```
    gen.scenario_set[0] = sce_add_sub;
```

```
    gen.start_xactor();
```

```
repeat(20) begin
```

```
    gen.out_chan.get(inst);
```

```
    inst.display();
```

```
end
```

```
end
```

```
endprogram
```

#### Down load the source code

[vmm\\_scenario.tar](#)

[Browse the code in vmm\\_scenario.tar](#)

### Command to simulate

```
vcs -sverilog -ntb_opts rvm -f filelist -R
```

Observe the log file, you can see the both the scenarios.

### Logfile

```
Instruction : LOAD__A | stream_id : 0 | scenario_id : 0
```

```
Instruction : LOAD__B | stream_id : 0 | scenario_id : 0
```

```
Instruction : ADD_A_B | stream_id : 0 | scenario_id : 0
```

```
Instruction : STORE_C | stream_id : 0 | scenario_id : 0
```

```
Instruction : LOAD__A | stream_id : 0 | scenario_id : 1
```

```
Instruction : LOAD__B | stream_id : 0 | scenario_id : 1
```

```
Instruction : ADD_A_B | stream_id : 0 | scenario_id : 1
```

```
Instruction : STORE_C | stream_id : 0 | scenario_id : 1
```

```
Instruction : LOAD__A | stream_id : 0 | scenario_id : 2
```

```
Instruction : LOAD__B | stream_id : 0 | scenario_id : 2
```

```
Instruction : SUB_A_B | stream_id : 0 | scenario_id : 2
```

```
Instruction : STORE_C | stream_id : 0 | scenario_id : 2
```

```
Instruction : LOAD__A | stream_id : 0 | scenario_id : 3
```

```
Instruction : LOAD__B | stream_id : 0 | scenario_id : 3
```

```
Instruction : ADD_A_B | stream_id : 0 | scenario_id : 3
```

```
Instruction : STORE_C | stream_id : 0 | scenario_id : 3
```

### VMM OPTS

Some of the test parameters in functional verification are passed from the command line. Passing Parameters from the command line allows simulating the same testcase file with different scenario. For example, let say you are verifying Ethernet protocol. Insert\_crc\_error.sv is a testcase file which verifies the DUT by sending CRC error packets. Now you can use the same testcase file to verify the DUT in 3 different modes , 10G 1000M and 100M modes by passing these parameters from the command prompt, this increases the testcase reusability.

As parameters became one of the most important techniques in reusing the testcases, vmm has added vmm\_opts class to manage parameters efficiently.

vmm\_opts captures parameter from run time options.

There are 2 ways to specify the parameters during runtime.

② 1) Command line

vmm\_opts internally uses \$plusargs to recognize the options specified on command line.

② 2) text file

If you have lot of parameters to mention, then you can also mention all the parameters in a text file and pass it as command line options.

vmm\_opts can recognize 3 types of parameters. They are

② String type parameters

② Integer type parameters

② Bit type parameters

Following are the 3 static methods which are defined in vmm\_opts to get the parameter values.

Static **function bit** get\_bit(**string name**, **string doc = ""**);

Static **function int** get\_int(**string name**, **int dflt = 0**, **string doc = ""**);

Static **function string** get\_string(**string name**, **string dflt = ""**, **string doc = ""**);

The first argument is the name of the parameter and the argument "doc" is description of the runtime argument that will be displayed by the vmm\_opts::get\_help() method. as these methods are static, user dont need to construct the object of vmm\_opts.

Argument "dflt" is the default value.

### How to give options

To supply a Boolean runtime option "foo" , use "+vmm\_opts+...+foo+..." command-line option, or "+vmm\_foo" command-line option, or the line "+foo" in the option file.

To supply a integer value "5" for runtime option "foo", use the "+vmm\_opts+...+foo=5+..." command-line option, or "+vmm\_foo=5" command-line option, or the line "+foo=5" in the option file.

To supply a string value "bar" for runtime option "foo" , use the "+vmm\_opts+...+foo=bar+..." command-line option, or "+vmm\_foo=bar" command-line option, or the line "+foo=bar" in the option file.

To supply a textfile which contains the runtime options, use "+vmm\_opts\_file=filename.txt" +vmm\_help command line option will print the run time options help. This option will call the vmm\_opts::get\_help() method.

The following is a basic example of how these parameters could be received using vmm\_opts

### EXAMPLE

```
`include "vmm.sv"
class my_env extends vmm_env;
    virtual function void gen_cfg();
        integer foo1;
        string foo2;
        bit foo3;

        super.gen_cfg();
        `vmm_note(log,"START OF GEN_CFG ");
        foo1 = vmm_opts::get_int("foo1" , 10 , "Help for the option foo1: pass any integer between 0 to 100");
        foo2 = vmm_opts::get_string("foo2" , "default_string" , "Help for the option foo2: pass any string");
        foo3 = vmm_opts::get_int("foo3" , "Help for the option foo3: just use foo3 to enable ");
        `vmm_note(log,$psprintf("\n  Run time Options \n    foo1 : %0d \n    foo2 : %s \n
foo3 : %b \n",foo1,foo2,foo3));
        `vmm_note(log,"END OF GEN_CFG ");
    endfunction
endclass
program main();
    initial
        begin
            my_env env;
            env = new();
            env.run();
        end
    endprogram
Download the example
vmm\_opts.tar
Browse the code in vmm\_opts.tar
```

### Commands to compile

This works with vmm 1.1 and above.

```
vcs -sverilog -ntb_opts dtm +incdir+$VMM_HOME/sv main_testcase.sv
```

### Commands to compile

```
./simv  
./simv +vmm_foo1=101  
./simv +vmm_opts+foo1=101+foo2=Testbench.in
```

Following is log file generated using runtime option +vmm\_foo1=101  
+vmm\_foo2=Testbench.in

### Log

Normal[NOTE] on Verif Env() at 0:

    START OF GEN\_CFG

WARNING[FAILURE] on vmm\_opts(class) at 0:

    No documentation specified for option "foo3".

Normal[NOTE] on Verif Env() at 0:

Run time Options

    foo1 : 101

    foo2 : Testbench.in

    foo3 : 0

Normal[NOTE] on Verif Env() at 0:

    END OF GEN\_CFG

Simulation PASSED on ./ (./) at 0 (1 warnings, 0 demoted errors & 0 demoted warnings)

## **INTRODUCTION**

The OVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog.

OVM library contains:

- ⌚ Component classes for building testbench components like generator/driver/monitor etc.
- ⌚ Reporting classes for logging,
- ⌚ Factory for object substitution.
- ⌚ Synchronization classes for managing concurrent process.
- ⌚ Policy classes for printing, comparing, recording, packing, and unpacking of ovm\_object

based classes.

- ⌚ TLM Classes for transaction level interface.
- ⌚ Sequencer and Sequence classes for generating realistic stimulus.
- ⌚ And Macros which can be used for shorthand notation of complex implementation.

In this tutorial, we will learn some of the OVM concepts with examples.

## **OVM TESTBENCH**

Ovm components, ovm env and ovm test are the three main building blocks of a testbench in ovm based verification.

### **Ovm\_env**

Ovm\_env is extended from ovm\_componented and does not contain any extra functionality. Ovm\_env is used to create and connect the ovm\_components like driver, monitors , sequencers etc. A environment class can also be used as sub-environment in another environment. As there is no difference between ovm\_env and ovm\_component , we will discuss about ovm\_component, in the next section.

### **Verification Components**

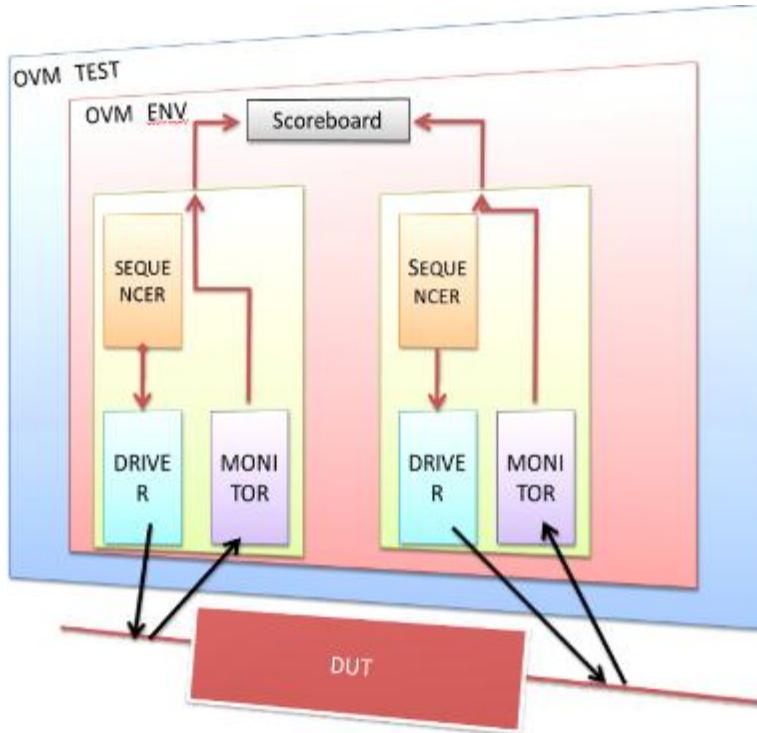
Ovm verification component classes are derived from ovm\_component class which provides features like hierarchy searching, phasing, configuration , reporting , factory and transaction recording.

Following are some of the ovm component classes

- ⌚ Ovm\_agent
- ⌚ Ovm\_monitor
- ⌚ Ovm\_scoreboard
- ⌚ Ovm\_driver
- ⌚ Ovm\_sequencer

NOTE: ovm\_env and ovm\_test are also extended from ovm\_component.

A typical ovm verification environment:



© www.testbench.in

An agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

#### About Ovm component Class:

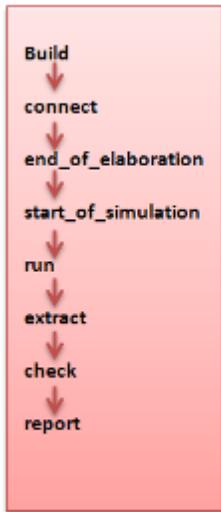
Ovm\_compoenent class is inherited from ovm\_report\_object which is inherited from ovm\_object.

As I mentioned previously, ovm\_component class provides features like hierarchy searching, phasing, configuration , reporting , factory and transaction recording.

We will discuss about phasing concept in this section and rest of the features will be discussed as separate topics.

#### OVM phases

OVM Components execute their behavior in strictly ordered, pre-defined phases. Each phase is defined by its own virtual method, which derived components can override to incorporate component-specific behavior. By default , these methods do nothing.



© www.bsoftnch.in

### --> **virtual function void** build()

This phase is used to construct various child components/ports/exports and configures them.

### --> **virtual function void** connect()

This phase is used for connecting the ports/exports of the components.

### --> **virtual function void** end\_of\_elaboration()

This phase is used for configuring the components if required.

### --> **virtual function void** start\_of\_simulation()

This phase is used to print the banners and topology.

### --> **virtual task** run()

In this phase , Main body of the test is executed where all threads are forked off.

### --> **virtual function void** extract()

In this phase, all the required information is gathered.

### --> **virtual function void** check()

In this phase, check the results of the extracted information such as un responded requests in

scoreboard, read statistics registers etc.

--> **virtual function void** report()

This phase is used for reporting the pass/fail status.

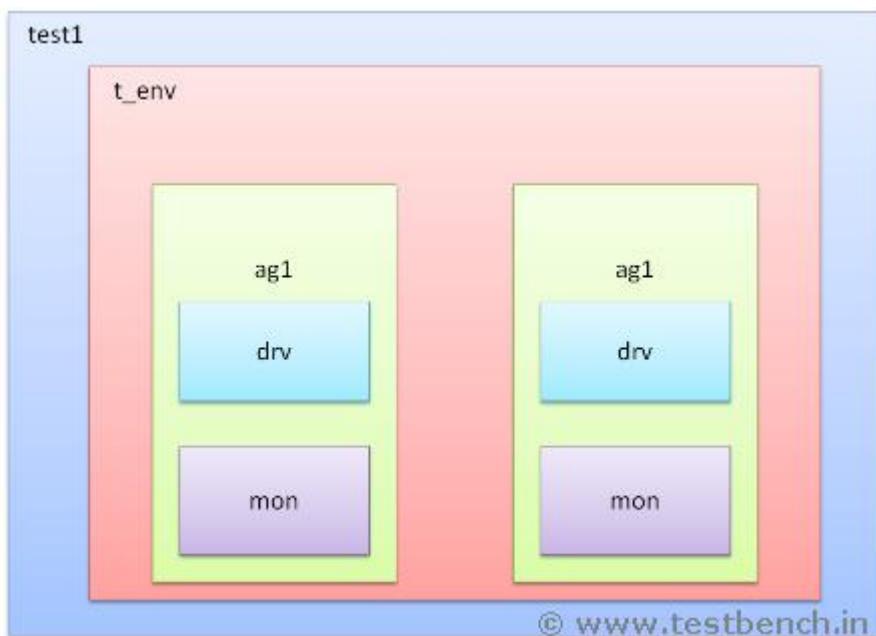
Only build() method is executed in top down manner. i.e after executing parent build() method, child objects build() methods are executed. All other methods are executed in bottom-up manner. The run() method is the only method which is time consuming. The run() method is forked, so the order in which all components run() method are executed is undefined.

### Ovm test

Ovm\_test is derived from ovm\_component class and there is no extra functionality is added. The advantage of used ovm\_test for defining the user defined test is that the test case selection can be done from command line option +OVM\_TESTNAME=<testcase\_string>. User can also select the testcase by passing the testcase name as string to ovm\_root::run\_test(<testcase\_string>) method.

In the above <testcase\_string> is the object type of the testcase class.

Lets implement environment for the following topology. I will describe the implementation of environment , testcase and top module. Agent, monitor and driver are implemented similar to environment.



1)Extend ovm\_env class and define user environment.

```
class env extends ovm_env;
```

2)Declare the utility macro. This utility macro provides the implementation of create() and get\_type\_name() methods and all the requirements needed for factory.

```
`ovm_component_utils(env)
```

3)Declare the objects for agents.

```
agent ag1;  
agent ag2;
```

4)Define the constructor. In the constructor, call the super methods and pass the parent object. Parent is the object in which environment is instantiated.

```
function new(string name , ovm_component parent = null);  
    super.new(name, parent);  
endfunction: new
```

5)Define build method. In the build method, construct the agents.

To construct agents, use create() method. The advantage of create() over new() is that when create() method is called, it will check if there is a factory override and constructs the object of override type.

```
function void build();  
    super.build();  
    ovm_report_info(get_full_name(),"Build", OVM_LOG);  
    ag1 = agent::type_id::create("ag1",this);  
    ag2 = agent::type_id::create("ag2",this);  
endfunction
```

6)Define connect(),end\_of\_elaboration(),start\_of\_simulation(),run(),extract(),check(),report() methods.

Just print a message from these methods, as we dont have any logic in this example to define.

```
function void connect();  
    ovm_report_info(get_full_name(),"Connect", OVM_LOG);  
endfunction
```

Complete code of environment class:

```
class env extends ovm_env;
```

```
`ovm_component_utils(env)  
agent ag1;  
agent ag2;
```

```

function new(string name, ovm_component parent);
    super.new(name, parent);
endfunction
function void build();
    ovm_report_info(get_full_name(),"Build", OVM_LOG);
    ag1 = agent::type_id::create("ag1",this);
    ag2 = agent::type_id::create("ag2",this);
endfunction
function void connect();
    ovm_report_info(get_full_name(),"Connect", OVM_LOG);
endfunction
function void end_of_elaboration();
    ovm_report_info(get_full_name(),"End_of_elaboration", OVM_LOG);
endfunction
function void start_of_simulation();
    ovm_report_info(get_full_name(),"Start_of_simulation", OVM_LOG);
endfunction
task run();
    ovm_report_info(get_full_name(),"Run", OVM_LOG);
endtask
function void extract();
    ovm_report_info(get_full_name(),"Extract", OVM_LOG);
endfunction
function void check();
    ovm_report_info(get_full_name(),"Check", OVM_LOG);
endfunction
function void report();
    ovm_report_info(get_full_name(),"Report", OVM_LOG);
endfunction
endclass

```

Now we will implement the testcase.

1)Extend ovm\_test and define the test case

```
class test1 extends ovm_test;
```

2)Declare component ustilts using utility macro.

```
ovm_component_utils(test1)
```

2)Declare environment class handle.

```
env t_env;
```

3)Define constructor method. In the constructor, call the super method and construct the environment object.

```

function new (string name="test1", ovm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new

```

4) Define the end\_of\_elaboration method. In this method, call the print() method. This print() method will print the topology of the test.

```

function void end_of_elaboration();
    ovm_report_info(get_full_name(),"End_of_elaboration", OVM_LOG);
    print();
endfunction

```

4) Define the run method and call the global\_stop\_request() method.

```

task run ();
    #1000;
    global_stop_request();
endtask : run

```

#### Testcase source code:

```

class test1 extends ovm_test;
`ovm_component_utils(test1)
 env t_env;
function new (string name="test1", ovm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new
function void end_of_elaboration();
    ovm_report_info(get_full_name(),"End_of_elaboration", OVM_LOG);
    print();
endfunction
task run ();
    #1000;
    global_stop_request();
endtask : run
endclass

```

## **Top Module:**

To start the testbench, run\_test() method must be called from initial block.  
Run\_test() method phases all components through all registered phases.

**module** top;

**initial**

    run\_test();

**endmodule**

[Download the source code](#)

[ovm\\_phases.tar](#)

[Browse the code in ovm\\_phases.tar](#)

[Command to run the simulation](#)

your\_tool\_simulation\_command +path\_to\_ovm\_pkg -f filelist +OVM\_TESTNAME=test1

[Log file:](#)

[RNTST] Running test test1...

```
ovm_test_top.t_env [ovm_test_top.t_env] Build
ovm_test_top.t_env.ag1 [ovm_test_top.t_env.ag1] Build
ovm_test_top.t_env.ag1.drv [ovm_test_top.t_env.ag1.drv] Build
ovm_test_top.t_env.ag1.mon [ovm_test_top.t_env.ag1.mon] Build
ovm_test_top.t_env.ag2 [ovm_test_top.t_env.ag2] Build
ovm_test_top.t_env.ag2.drv [ovm_test_top.t_env.ag2.drv] Build
ovm_test_top.t_env.ag2.mon [ovm_test_top.t_env.ag2.mon] Build
ovm_test_top.t_env.ag1.drv [ovm_test_top.t_env.ag1.drv] Connect
ovm_test_top.t_env.ag1.mon [ovm_test_top.t_env.ag1.mon] Connect
ovm_test_top.t_env.ag1 [ovm_test_top.t_env.ag1] Connect
ovm_test_top.t_env.ag2.drv [ovm_test_top.t_env.ag2.drv] Connect
ovm_test_top.t_env.ag2.mon [ovm_test_top.t_env.ag2.mon] Connect
ovm_test_top.t_env.ag2 [ovm_test_top.t_env.ag2] Connect
ovm_test_top.t_env [ovm_test_top.t_env] Connect
ovm_test_top.t_env.ag1.drv [ovm_test_top.t_env.ag1.drv] End_of_elaboration
ovm_test_top.t_env.ag1.mon [ovm_test_top.t_env.ag1.mon] End_of_elaboration
ovm_test_top.t_env.ag1 [ovm_test_top.t_env.ag1] End_of_elaboration
ovm_test_top.t_env.ag2.drv [ovm_test_top.t_env.ag2.drv] End_of_elaboration
ovm_test_top.t_env.ag2.mon [ovm_test_top.t_env.ag2.mon] End_of_elaboration
ovm_test_top.t_env.ag2 [ovm_test_top.t_env.ag2] End_of_elaboration
ovm_test_top.t_env [ovm_test_top.t_env] End_of_elaboration
ovm_test_top [ovm_test_top] End_of_elaboration
```

---

Name	Type	Size	Value
------	------	------	-------

---

```

ovm_test_top      test1      -      ovm_test_top@2
t_env            env        -      t_env@4
ag1              agent      -      ag1@6
drv              driver     -      drv@12
rsp_port         ovm_analysis_port -      rsp_port@16
sqr_pull_port   ovm_seq_item_pull_+ -      sqr_pull_port@14
mon              monitor    -      mon@10
ag2              agent      -      ag2@8
drv              driver     -      drv@20
rsp_port         ovm_analysis_port -      rsp_port@24
sqr_pull_port   ovm_seq_item_pull_+ -      sqr_pull_port@22
mon              monitor    -      mon@18

```

```

ovm_test_top.t_env.ag1.drv[ovm_test_top.t_env.ag1.drv]Start_of_simulation
ovm_test_top.t_env.ag1.mon[ovm_test_top.t_env.ag1.mon]Start_of_simulation
ovm_test_top.t_env.ag1[ovm_test_top.t_env.ag1]Start_of_simulation
ovm_test_top.t_env.ag2.drv[ovm_test_top.t_env.ag2.drv]Start_of_simulation
ovm_test_top.t_env.ag2.mon[ovm_test_top.t_env.ag2.mon]Start_of_simulatio

```

```

..
..
..
..
```

Observe the above log report:

1)Build method was called in top-down fashion. Look at the following part of message.

```

ovm_test_top.t_env [ovm_test_top.t_env] Build
ovm_test_top.t_env.ag1 [ovm_test_top.t_env.ag1] Build
ovm_test_top.t_env.ag1.drv [ovm_test_top.t_env.ag1.drv] Build
ovm_test_top.t_env.ag1.mon [ovm_test_top.t_env.ag1.mon] Build
ovm_test_top.t_env.ag2 [ovm_test_top.t_env.ag2] Build
ovm_test_top.t_env.ag2.drv [ovm_test_top.t_env.ag2.drv] Build
ovm_test_top.t_env.ag2.mon [ovm_test_top.t_env.ag2.mon] Build

```

2)Connect method was called in bottom up fashion. Look at the below part of log file,

```

ovm_test_top.t_env.ag1.drv [ovm_test_top.t_env.ag1.drv] Connect
ovm_test_top.t_env.ag1.mon [ovm_test_top.t_env.ag1.mon] Connect
ovm_test_top.t_env.ag1 [ovm_test_top.t_env.ag1] Connect
ovm_test_top.t_env.ag2.drv [ovm_test_top.t_env.ag2.drv] Connect
ovm_test_top.t_env.ag2.mon [ovm_test_top.t_env.ag2.mon] Connect
ovm_test_top.t_env.ag2 [ovm_test_top.t_env.ag2] Connect
ovm_test_top.t_env [ovm_test_top.t_env] Connect

```

3)Following part of log file shows the testcase topology.

Name	Type	Size	Value
ovm_test_top	test1	-	ovm_test_top@2
t_env	env	-	t_env@4
ag1	agent	-	ag1@6
drv	driver	-	drv@12
rsp_port	ovm_analysis_port	-	rsp_port@16
sqr_pull_port	ovm_seq_item_pull_*	-	sqr_pull_port@14
mon	monitor	-	mon@10
ag2	agent	-	ag2@8
drv	driver	-	drv@20
rsp_port	ovm_analysis_port	-	rsp_port@24
sqr_pull_port	ovm_seq_item_pull_*	-	sqr_pull_port@22
mon	monitor	-	mon@10

## OVM REPORTING

The ovm\_report\_object provides an interface to the OVM reporting facility. Through this interface, components issue the various messages with different severity levels that occur during simulation. Users can configure what actions are taken and what file(s) are output for individual messages from a particular component or for all messages from all components in the environment.

A report consists of an id string, severity, verbosity level, and the textual message itself. If the verbosity level of a report is greater than the configured maximum verbosity level of its report object, it is ignored.

### Reporting Methods:

Following are the primary reporting methods in the OVM.

**virtual function void** ovm\_report\_info

**(string id,string message,int verbosity=OVM\_MEDIUM,string filename="",int line=0)**

**virtual function void** ovm\_report\_warning

**(string id,string message,int verbosity=OVM\_MEDIUM,string filename="",int line=0)**

**virtual function void** ovm\_report\_error

**(string id,string message,int verbosity=OVM\_LOW, string filename="",int line=0)**

**virtual function void** ovm\_report\_fatal

**(string id,string message,int verbosity=OVM\_NONE, string filename="",int line=0)**

### Arguments description:

- ⌚ id -- a unique id to form a group of messages.
- ⌚ message -- The message text
- ⌚ verbosity -- the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level, then the report is issued, subject to the configured action and file descriptor settings.
- ⌚ filename/line -- If required to print filename and line number from where the message is issued, use macros, `\_\_FILE\_\_` and `\_\_LINE\_\_`.

#### **Actions:**

These methods associate the specified action or actions with reports of the given severity, id, or severity-id pair.

Following are the actions defined:

- ⌚ OVM\_NO\_ACTION -- Do nothing
- ⌚ OVM\_DISPLAY -- Display report to standard output
- ⌚ OVM\_LOG -- Write to a file
- ⌚ OVM\_COUNT -- Count up to a max\_quit\_count value before exiting
- ⌚ OVM\_EXIT -- Terminates simulation immediately
- ⌚ OVM\_CALL\_HOOK -- Callback the hook method .

#### **Configuration:**

Using these methods, user can set the verbosity levels and set actions.

```
function void set_report_verbosity_level
    (int verbosity_level)
function void set_report_severity_action
    (ovm_severity severity, ovm_action action)
function void set_report_id_action
    (string id, ovm_action action)
function void set_report_severity_id_action
    (ovm_severity severity, string id, ovm_action action)
```

OVM Reporting API		
	Types	Methods
<b>Severity</b>	OVM_INFO	ovm_report_info
	OVM_WARNING	ovm_report_warning
	OVM_ERROR	ovm_report_error
	OVM_FATAL	ovm_report_fatal
<b>Filtering</b>	set_report_verbosity_level	
<b>Actions</b>	Types	Methods
	OVM_NO_ACTION	set_report_id_action
	OVM_DISPLAY	
	OVM_LOG	set_report_severity_action
	OVM_COUNT	
	OVM_EXIT	set_report_severity_id_action
<b>Outputfile</b>	set_report_default_file	
	set_report_severity_file	
	set_report_id_file	
	set_report_severity_id_file	
<b>Query</b>	get_report_verbosity_level	
	get_report_action	
	get_report_file_handle	

© www.testbench.in

### Example

Lets see an example:

In the following example, messages from rpting::run() method are of different verbosity level. In the top module, 3 objects of rpting are created and different verbosity levels are set using set\_report\_verbosity\_level() method.

```
`include "ovm.svh"
import ovm_pkg::*;

class rpting extends ovm_threaded_component;

`ovm_component_utils(rpting)
function new(string name, ovm_component parent);
    super.new(name, parent);
endfunction

task run();
    ovm_report_info(get_full_name(),
        "Info Message : Verbo lvl - OVM_NONE ", OVM_NONE, `__FILE__, `__LINE__);
    ovm_report_info(get_full_name(),
        "Info Message : Verbo lvl - OVM_LOW ", OVM_LOW);
    ovm_report_info(get_full_name(),
        "Info Message : Verbo lvl - 150 ", 150);
endtask
```

```

ovm_report_info(get_full_name(),
"Info Message : Verbo lvl - OVM_MEDIUM",OVM_MEDIUM);

ovm_report_warning(get_full_name(),
"Warning Messgae from rpting",OVM_LOW);

ovm_report_error(get_full_name(),
"Error Message from rpting \n\n",OVM_LOG);
endtask
endclass

module top;
rpting rpt1;
rpting rpt2;
rpting rpt3;
initial begin
rpt1 = new("rpt1",null);
rpt2 = new("rpt2",null);
rpt3 = new("rpt3",null);
rpt1.set_report_verbosity_level(OVM_MEDIUM);
rpt2.set_report_verbosity_level(OVM_LOW);
rpt3.set_report_verbosity_level(OVM_NONE);
run_test();
end
endmodule

```

[Download the source code](#)

[ovm\\_reporting.tar](#)

[Browse the code in ovm\\_reporting.tar](#)

[Command to run the simulation](#)

your\_tool\_simulation\_command +path\_to\_ovm\_pkg ovm\_log\_example.sv

[Log file:](#)

```

OVM_INFO reporting.sv(13)@0:rpt1[rpt1]Info Message:Verbo lvl - OVM_NONE
OVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - OVM_LOW
OVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - 150
OVM_INFO @0:rpt1[rpt1] Info Message : Verbo lvl - OVM_MEDIUM
OVM_WARNIN@0:rpt[rpt1] Warning Messgae from rpting
OVM_ERROR @0:rpt1[rpt1] Error Message from rpting
OVM_INFOrouting.sv(13)@0:rpt2[rpt2]Info Message:Verbo lvl - OVM_NONE
OVM_INFO@ 0:rpt2[rpt2] Info Message : Verbo lvl - OVM_LOW
OVM_WARNING@0:rpt2[rpt2] Warning Messgae from rpting
OVM_ERROR@0:rpt2[rpt2] Error Message from rpting

```

OVM\_INFOrouting.sv(13)@0:rpt3[rpt3]Info Message:Verbo lvl - OVM\_NONE  
OVM\_ERROR @ 9200 [TIMOUT] Watchdog timeout of '23f0' expired.

## **OVM TRANSACTION**

A transaction is data item which is eventually or directly processed by the DUT. The packets, instructions, pixels are data items. In ovm, transactions are extended from ovm\_transactions class or ovm\_sequence\_item class. Generally transactions are extended from ovm\_transaction if randomization is not done in sequence and transactions are extended from ovm\_sequence\_item if the randomization is done in sequence. In this section, we will see ovm\_transaction only, ovm\_sequence\_item will be addressed in another section.

### **Example of a transaction:**

```
class Packet extends ovm_transaction;  
    rand bit [7:0] da;  
    rand bit [7:0] sa;  
    rand bit [7:0] length;  
    rand bit [7:0] data[];  
    rand byte fcs;  
endclass
```

### **Core Utilities:**

ovm\_transaction class is extended from ovm\_object. Ovm\_transaction adds more features like transaction recording , transaction id and timings of the transaction.

The methods used to model, transform or operate on transactions like print, copying, cloning, comparing, recording, packing and unpacking are already defined in ovm\_object.

<b>Utilities Method</b>	<b>Description</b>
print	Prints this object's properties
record	Records this object's properties
copy	Returns a deep copy of this object.
compare	Deep compares this data object with the object provided in the rhs (right-hand side) argument.
pack	This methods bitwise-concatenate this object's properties into an array
unpack	Extract property values from an array
clone	Clone method creates and returns an exact copy of this object. The default implementation calls create method followed by copy method.
create	This method allocates a new object of the same type as this object and returns it via a base ovm_object handle.

© [www.testbench.in](http://www.testbench.in)

FIG: OVM OBJECT UTILITIES

### User Defined Implementations:

User should define these methods in the transaction using do\_<method\_name> and call them using <method\_name>. Following table shows calling methods and user-defined hook do\_<method\_name> methods. Clone and create methods, does not use hook methods concepts.

Functionality	User defined hook methods	Calling methods
Printing	do_print	print
		sprint
		convert2string
Recording	do_record	record
Copying	do_copy	copy
Comparing	do_compare	compare
Packing	do_pack	pack
		pack_bytes
		pack_ints
Unpacking	do_unpack	unpack
		unpack_bytes
		unpack_ints
Cloning	clone	clone
Create	create	create

© www.testbench.in

### Shorthand Macros:

Using the field automation concept of ovm, all the above defines methods can be defined automatically.

To use these field automation macros, first declare all the data fields, then place the field automation macros between the `ovm\_object\_utils\_begin and `ovm\_object\_utils\_end macros.

### Example of field automation macros:

```
class Packet extends ovm_transaction;
```

```

rand bit [7:0] da;
rand bit [7:0] sa;
rand bit [7:0] length;
rand bit [7:0] data[];
rand byte    fcs;

`ovm_object_utils_begin(Packet)
  `ovm_field_int(da, OVM_ALL_ON|OVM_NOPACK)
  `ovm_field_int(sa, OVM_ALL_ON|OVM_NOPACK)
  `ovm_field_int(length, OVM_ALL_ON|OVM_NOPACK)
```

```

`ovm_field_array_int(data, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_int(fcs, OVM_ALL_ON|OVM_NOPACK)
`ovm_object_utils_end

```

**endclass.**

For most of the data types in systemverilog, ovm defined corresponding field automation macros. Following table shows all the field automation macros.

Type	Macros
Scalar	'ovm_field_int 'ovm_field_object 'ovm_field_string 'ovm_field_enum 'ovm_field_real 'ovm_field_event
Static Array	'ovm_field_sarray_int 'ovm_field_sarray_object 'ovm_field_sarray_string 'ovm_field_sarray_enum
Dynamic array	'ovm_field_array_int 'ovm_field_array_object 'ovm_field_array_string 'ovm_field_array_enum
Queue	'ovm_field_queue_int 'ovm_field_queue_object 'ovm_field_queue_string 'ovm_field_queue_enum
Associative array with string index	'ovm_field_aa_int_string 'ovm_field_aa_object_string 'ovm_field_aa_string_string
Associative array with integral index	'ovm_field_aa_object_int 'ovm_field_aa_int_int 'ovm_field_aa_int_int_unsigned 'ovm_field_aa_int_integer 'ovm_field_aa_int_integer_unsigned 'ovm_field_aa_int_byte 'ovm_field_aa_int_byte_unsigned 'ovm_field_aa_int_shortint 'ovm_field_aa_int_shortint_unsigned 'ovm_field_aa_int_longint 'ovm_field_aa_int_longint_unsigned 'ovm_field_aa_int_key 'ovm_field_aa_int_enumkey

Each `ovm\_field\_\*` macro has at least two arguments: ARG and FLAG.

ARG is the instance name of the variable and FLAG is used to control the field usage in core utilities operation.

Following table shows ovm field automation flags:

FLAG	DESCRIPTION
OVM_ALL_ON	Set all operations
OVM_DEFAULT	Use the default flag settings.
OVM_NOCOPY	Do not copy this field.
OVM_NOCOMPARE	Do not compare this field.
OVM_NOPRINT	Do not print this field.
OVM_NODEFPRINT	Do not print the field if it is the same as its
OVM_NOPACK	Do not pack or unpack this field.
OVM_PHYSICAL	Treat as a physical field.
OVM_ABSTRACT	Treat as an abstract field.
OVM_READONLY	Do not allow setting of this field from the set_*_local methods.

© [www.testbench.in](http://www.testbench.in)

By default, FLAG is set to OVM\_ALL\_ON. All these flags can be ored. Using NO\_\* flags, can turn off particular field usage in a paerticular method. NO\_\* flags takes precedence over other flags.

#### Example of Flags:

```
`ovm_field_int(da, OVM_ALL_ON|OVM_NOPACK)
```

The above macro will use the field "da" in all utilities methods except Packing and unpacking methods.

#### Lets see a example:

In the following example, all the utility methods are defined using field automation macros except Packing and unpacking methods. Packing and unpacking methods are done in do\_pack() and do\_unpack() method.

```
`include "ovm.svh"  
import ovm_pkg::*;  
//Define the enumerated types for packet types  
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;  
class Packet extends ovm_transaction;
```

```

rand fcs_kind_t    fcs_kind;

rand bit [7:0] length;
rand bit [7:0] da;
rand bit [7:0] sa;
rand bit [7:0] data[];
rand byte fcs;
constraint payload_size_c { data.size inside { [1 : 6]};}

constraint length_c { length == data.size; }

constraint solve_size_length { solve data.size before length; }

function new(string name = "");
    super.new(name);
endfunction : new

function void post_randomize();
    if(fcs_kind == GOOD_FCS)
        fcs = 8'b0;
    else
        fcs = 8'b1;
    fcs = cal_fcs();
endfunction : post_randomize

<:/// method to calculate the fcs /////
virtual function byte cal_fcs;
    integer i;
    byte result ;
    result = 0;
    result = result ^ da;
    result = result ^ sa;
    result = result ^ length;
    for (i = 0;i< data.size;i++)
        result = result ^ data[i];
    result = fcs ^ result;
    return result;
endfunction : cal_fcs

`ovm_object_utils_begin(Packet)

```

```

`ovm_field_int(da, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_int(sa, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_int(length, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_array_int(data, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_int(fcs, OVM_ALL_ON|OVM_NOPACK)
`ovm_object_utils_end

function void do_pack(ovm_packer packer);
    super.do_pack(packer);
    packer.pack_field_int(da,$bits(da));
    packer.pack_field_int(sa,$bits(sa));
    packer.pack_field_int(length,$bits(length));
    foreach(data[i])
        packer.pack_field_int(data[i],8);
    packer.pack_field_int(fcs,$bits(fcs));
endfunction : do_pack

function void do_unpack(ovm_packer packer);
    int sz;
    super.do_pack(packer);

    da = packer.unpack_field_int($bits(da));
    sa = packer.unpack_field_int($bits(sa));
    length = packer.unpack_field_int($bits(length));

    data.delete();
    data = new[length];
    foreach(data[i])
        data[i] = packer.unpack_field_int(8);
    fcs = packer.unpack_field_int($bits(fcs));
endfunction : do_unpack

endclass : Packet
///////////
/// Test to check the packet implementation ///
///////////

module test;

Packet pkt1 = new("pkt1");
Packet pkt2 = new("pkt2");

```

```

byte unsigned pkdbytes[];
```

```

initial
repeat(10)
  if(pkt1.randomize)
    begin
      $display(" Randomization Sucessesfull.");
      pkt1.print();
      ovm_default_packer.use_metadata = 1;
      void(pkt1.pack_bytes(pkdbytes));
      $display("Size of pkd bits %d",pkdbytes.size());
      pkt2.unpack_bytes(pkdbytes);
      pkt2.print();
      if(pkt2.compare(pkt1))
        $display(" Packing,Unpacking and compare worked");
      else
        $display(" *** Something went wrong in Packing or Unpacking or compare *** \n \n");
    end
    else
      $display(" *** Randomization Failed ***");

```

**endmodule**

[Download the source code](#)

[ovm\\_transaction.tar](#)

[Browse the code in ovm\\_transaction.tar](#)

[Command to run the simulation](#)

your\_tool\_simulation\_command +path\_to\_ovm\_pkg -packet.sv

[Log report:](#)

Randomization Sucessesfull.

Name	Type	Size	Value
pkt1	Packet	-	pkt1@3
da	integral	8	'h1d
sa	integral	8	'h26
length	integral	8	'h5

data	da(integral)	5	-
[0]	integral	8	'hb1
[1]	integral	8	'h3f
[2]	integral	8	'h9e
[3]	integral	8	'h38
[4]	integral	8	'h8d
fcs	integral	8	'h9b
<hr/>			
Size of pkd bits	9		
<hr/>			
Name	Type	Size	Value
pkt2	Packet	-	pkt2@5
da	integral	8	'h1d
sa	integral	8	'h26
length	integral	8	'h5
data	da(integral)	5	-
[0]	integral	8	'hb1
[1]	integral	8	'h3f
[2]	integral	8	'h9e
[3]	integral	8	'h38
[4]	integral	8	'h8d
fcs	integral	8	'h9b

Packing,Unpacking and compare worked

## **OVM FACTORY**

The factory pattern is an well known object-oriented design pattern. The factory method design pattern defining a separate method for creating the objects. , whose subclasses can then override to specify the derived type of object that will be created.

Using this method, objects are constructed dynamically based on the specification type of the object. User can alter the behavior of the pre-build code without modifying the code. From the testcase, user from environment or testcase can replace any object which is at any hierarchy level with the user defined object.

For example: In your environment, you have a driver component. You would like the extend the driver component for error injection scenario. After defining the extended driver class with error injection, how will you replace the base driver component which is deep in the hierarchy of your environment ? Using hierarchical path, you could replace the driver object with the extended driver. This could not be easy if there are many driver objects. Then you should also take care of its connections with the other components of testbenches like scoreboard etc.

One more example: In your Ethernet verification environment, you have different drivers to support different interfaces for 10mbps,100mps and 1G. Now you want to reuse the same

environment for 10G verification. Inside somewhere deep in the hierarchy, while building the components, as a driver components ,your current environment can only select 10mmmps/100mps/1G drivers using configuration settings. How to add one more driver to the current drivers list of drivers so that from the testcase you could configure the environment to work for 10G.

Using the ovm fatroy, it is very easy to solve the above two requirements. Only classss extended from ovm\_object and ovm\_component are supported for this.

There are three basic steps to be followed for using ovm factory.

- ⌚ 1) Registration
- ⌚ 2) Construction
- ⌚ 3) Overriding

The factory makes it is possible to override the type of ovm component /object or instance of a ovm component/object in2 ways. They are based on ovm component/object type or ovm componenent/object name.

### **Registration:**

While defining a class , its type has to be registered with the ovm factory. To do this job easier, ovm has predefined macros.

```
`ovm_component_utils(class_type_name)
`ovm_component_param_utils(class_type_name #(params))
`ovm_object_utils(class_type_name)
`ovm_object_param_utils(class_type_name #(params))
```

For ovm\_\*\_param\_utils are used for parameterized classes and other two macros for non-parameterized class. Registration is required for name-based overriding , it is not required for type-based overriding.

### **EXAMPLE: Example of above macros**

```
class packet extends ovm_object;
  `ovm_object_utils(packet)
endclass
```

```
class packet #(type T=int, int mode=0) extends ovm_object;
  `ovm_object_param_utils(packet #(T,mode))
endclass
```

```
class driver extends ovm_component;
  `ovm_component_utils(driver)
```

```
endclass
```

```
class monitor #(type T=int, int mode=0) extends ovm_component;  
  `ovm_component_param_utils(driver#(T,mode))
```

```
endclass
```

### Construction:

To construct a ovm based component or ovm based objects, static method create() should be used. This function constructs the appropriate object based on the overrides and constructs the object and returns it. So while constructing the ovm based components or ovm based objects , do not use new() constructor.

Syntax :

```
static function T create(string name,  
                         ovm_component parent,  
                         string context = " ")
```

The Create() function returns an instance of the component type, T, represented by this proxy, subject to any factory overrides based on the context provided by the parents full name. The context argument, if supplied, supersedes the parents context. The new instance will have the given leaf name and parent.

### **EXAMPLE:**

```
class_type object_name;
```

```
object_name = class_type::type_id::creat("object_name",this);
```

For ovm\_object based classes, doesnt need the parent handle as second argument.

### Overriding:

If required, user could override the registered classes or objects. User can override based of name string or class-type.

There are 4 methods defined for overriding:

```
function void set_inst_override_by_type  
  (ovm_object_wrapper original_type,  
   ovm_object_wrapper override_type,  
   string full_inst_path )
```

The above method is used to override the object instances of "original\_type" with "override\_type" . "override\_type" is extended from"original\_type".

```
function void set_inst_override_by_name  
  (string original_type_name,  
   string override_type_name,  
   string full_inst_path )
```

Original\_type\_name and override\_type\_name are the class names which are registered in the factory. All the instances of objects with name "Original\_type\_name" will be overriden with objects of name "override\_type\_name" using set\_inst\_override\_by\_name() method.

```
function void set_type_override_by_type  
    (ovm_object_wrapper original_type,  
     ovm_object_wrapper override_type,  
     bit replace = 1)
```

Using the above method, request to create an object of original\_type can be overriden with override\_type.

```
function void set_type_override_by_name  
    (string original_type_name,  
     string override_type_name,  
     bit replace = 1)
```

Using the above method, request to create an object of original\_type\_name can be overriden with override\_type\_name.

When multiple overrides are done , then using the argument "replace" , we can control whether to override the previous override or not. If argument "replace" is 1, then previous overrides will be replaced otherwise, previous overrides will remain.

print() method, prints the state of the ovm\_factory, registered types, instance overrides, and type overrides.

Now we will see a complete example. This example is based on the environment build in topic OVM TESTBENCH . Refer to that section for more information about this example.

Lets look at the 3 steps which I discussed above using the example defined in OVM TESTBENCH

① Registration

In all the class, you can see the macro `ovm\_component\_utils(type\_name)

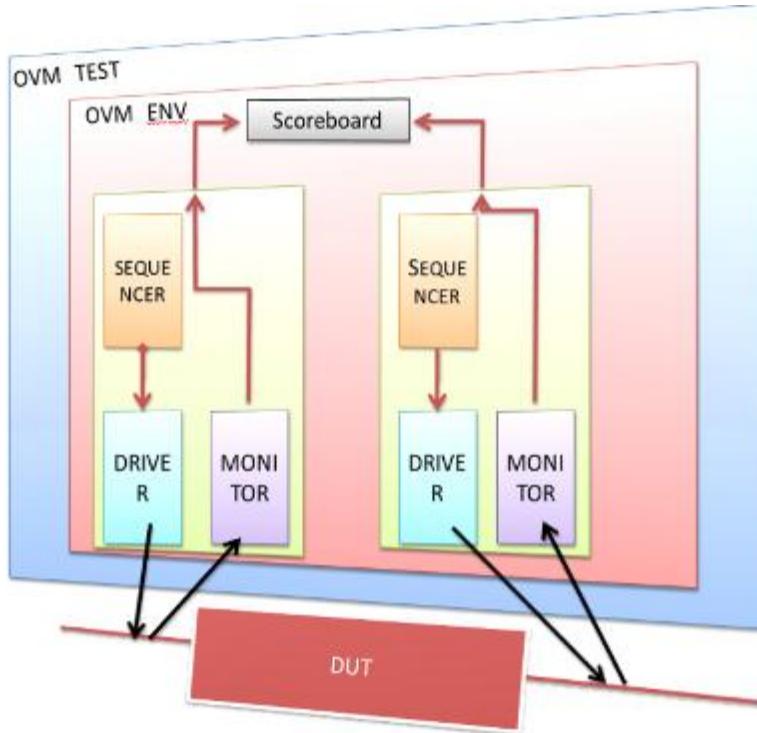
② Construction

In file agant.sv file, monitor and driver are constructed using create() method.

```
mon = monitor::type_id::create("mon",this);  
drv = driver::type_id::create("drv",this);
```

③ In this example, a one testcase is already developed in topic OVM\_TESTBENCH. There are no over rides in this test case.

Topology of this test environment is shown below.



© www.testbench.in

In this example, there is one driver class and one monitor class. In this testcase , By extending driver class , we will define driver\_2 class and by extending monitor class, we will define monitor\_2 class.

From the testcase , Using set\_type\_override\_by\_type, we will override driver with driver\_2 and Using set\_type\_override\_by\_name, we will override monitor with monitor\_2.

To know about the overrides which are done, call print\_all\_overrides() method of factory class.  
**class** driver\_2 **extends** driver;

```

`ovm_component_utils(driver_2)

function new(string name, ovm_component parent);
    super.new(name, parent);
endfunction
endclass

class monitor_2 extends monitor;

`ovm_component_utils(monitor_2)

function new(string name, ovm_component parent);
    super.new(name, parent);
endfunction
endclass
```

```

class test_factory extends ovm_test;

`ovm_component_utils(test_factory)
env t_env;

function new (string name="test1", ovm_component parent=null);
    super.new (name, parent);

    factory.set_type_override_by_type(driver::get_type(),driver_2::get_type(),"*");
    factory.set_type_override_by_name("monitor","monitor_2","");
    factory.print_all_overrides();
    t_env = new("t_env",this);
endfunction : new

function void end_of_elaboration();
    ovm_report_info(get_full_name(),"End_of_elaboration", OVM_LOG);
    print();
endfunction : end_of_elaboration
task run ();
    #1000;
    global_stop_request();
endtask : run

endclass

```

[Download the example:](#)

[ovm\\_factory.tar](#)

[Browse the code in ovm\\_factory.tar](#)

[Command to simulate](#)

Command to run the example with the testcase which is defined above:  
Your\_tool\_simulation\_command +incdir+path\_to\_ovm -f filelist  
+OVM\_TESTNAME=test\_factory

Method factory.print\_all\_overrides() displayed all the overrides as shown below in the log file.

#### Factory Configuration (\*)

No instance overrides are registered with this factory

Type Overrides:

Requested Type    Override Type

-----  
driver        driver\_2

monitor	monitor_2
---------	-----------

In the below text printed by print\_topology() method ,we can see overridden driver and monitor.

Name	Type	Size	Value
ovm_test_top	test_factory	-	ovm_test_top@2
t_env	env	-	t_env@4
ag1	agent	-	ag1@6
drv	driver_2	-	drv@12
rsp_port	ovm_analysis_port	-	rsp_port@16
sqr_pull_port	ovm_seq_item_pull_+ -	-	sqr_pull_port@14
mon	monitor_2	-	mon@10
ag2	agent	-	ag2@8
drv	driver_2	-	drv@20
rsp_port	ovm_analysis_port	-	rsp_port@24
sqr_pull_port	ovm_seq_item_pull_+ -	-	sqr_pull_port@22
mon	monitor_2	-	mon@18

In the below text printed by print\_topology() method ,with testcase test1 which does note have overrides.

Command to run this example with test1 is  
your\_tool\_simulation\_command +inccdir+path\_to\_ovm -f filelist +OVM\_TESTNAME=test1

Name	Type	Size	Value
ovm_test_top	test1	-	ovm_test_top@2
t_env	env	-	t_env@4
ag1	agent	-	ag1@6
drv	driver	-	drv@12
rsp_port	ovm_analysis_port	-	rsp_port@16
sqr_pull_port	ovm_seq_item_pull_+ -	-	sqr_pull_port@14
mon	monitor	-	mon@10
ag2	agent	-	ag2@8

```

drv      driver      -      drv@20
rsp_port   ovm_analysis_port -      rsp_port@24
sqr_pull_port  ovm_seq_item_pull_+ -  sqr_pull_port@22
mon      monitor      -      mon@18

```

## OVM SEQUENCE 1

### Introduction

A sequence is a series of transaction. User can define the complex stimulus. sequences can be reused, extended, randomized, and combined sequentially and hierarchically in various ways.

For example, for a processor, lets say PUSH\_A,PUSH\_B,ADD,SUB,MUL,DIV and POP\_C are the instructions. If the instructions are generated randomly, then to excusing a meaningful operation like "adding 2 variables" which requires a series of transaction

"PUSH\_A PUSH\_B ADD POP\_C " will take longer time. By defining these series of "PUSH\_A PUSH\_B ADD POP\_C ", it would be easy to exercise the DUT.

Advantages of ovm sequences :

- ⌚ Sequences can be reused.
- ⌚ Stimulus generation is independent of testbench.
- ⌚ Easy to control the generation of transaction.
- ⌚ Sequences can be combined sequentially and hierarchically.

A complete sequence generation requires following 4 classes.

1- Sequence item.

2- Sequence

3- Sequencer

4- Driver

- ⌚ Ovm\_sequence\_item :

User has to define a transaction by extending ovm\_sequence\_item. ovm\_sequence\_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism. For more information about ovm\_sequence\_item Refer to link

## OVM TRANSACTION

- ⌚ Ovm\_sequence:

User should extend ovm\_sequence class and define the construction of sequence of transactions. These transactions can be directed, constrained randomized or fully randomized. The ovm\_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

**virtual class ovm\_sequence #(**

**type REQ = ovm\_sequence\_item,**

**type RSP = REQ**

**)**

© Ovm\_sequencer:

Ovm\_sequencer is responsible for the coordination between sequence and driver. Sequencer sends the transaction to driver and gets the response from the driver. The response transaction from the driver is optional. When multiple sequences are running in parallel, then sequencer is responsible for arbitrating between the parallel sequences. There are two types of sequencers : ovm\_sequencer and ovm\_push\_sequencer

```
class ovm_sequencer #(  
    type REQ = ovm_sequence_item,  
    type RSP = REQ  
)  
  
class ovm_push_sequencer #(  
    type REQ = ovm_sequence_item,  
    type RSP = REQ  
)
```

© Ovm driver:

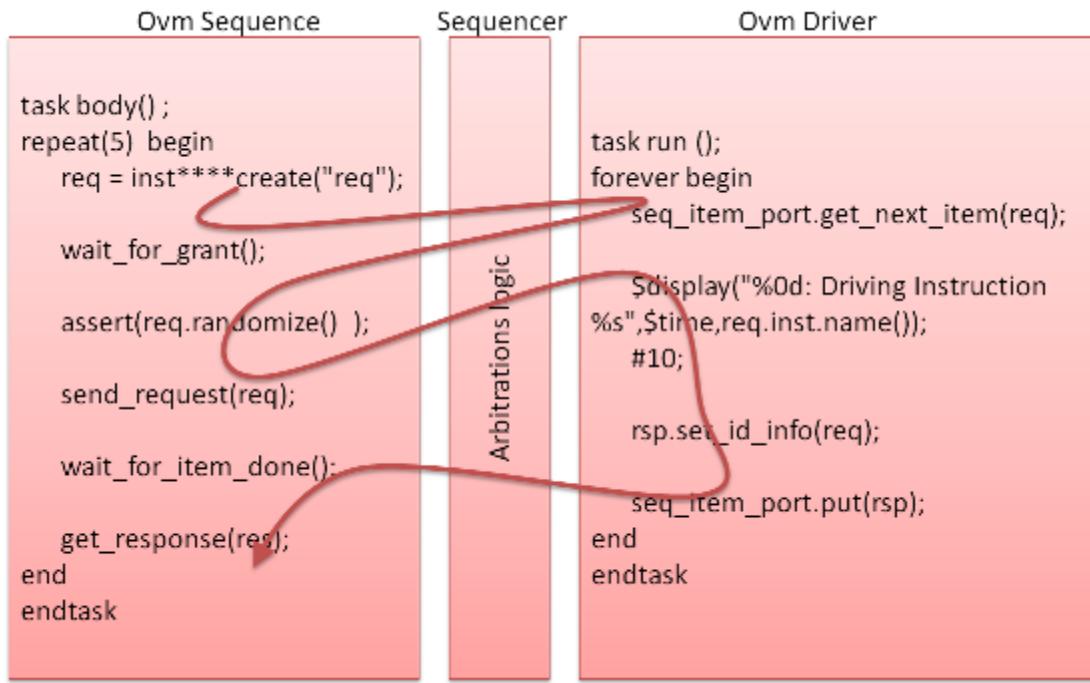
User should extend ovm\_driver class to define driver component. ovm driver is a component that initiate requests for new transactions and drives it to lower level components. There are two types of drivers: ovm\_driver and ovm\_push\_driver.

```
class ovm_driver #(  
    type REQ = ovm_sequence_item,  
    type RSP = REQ  
)  
  
class ovm_push_driver #(  
    type REQ = ovm_sequence_item,  
    type RSP = REQ  
)
```

In pull mode , ovm\_sequencer is connected to ovm\_driver , in push mode ovm\_push\_sequencer is connectd to ovm\_push\_driver.

Ovm\_sequencer and ovm\_driver are parameterized components with request and response transaction types. REQ and RSP types by default are ovm\_sequence\_type types. User can specify REQ and RSP of different transaction types. If user specifies only REQ type, then RSP will be REQ type.

## Sequence And Driver Communication:

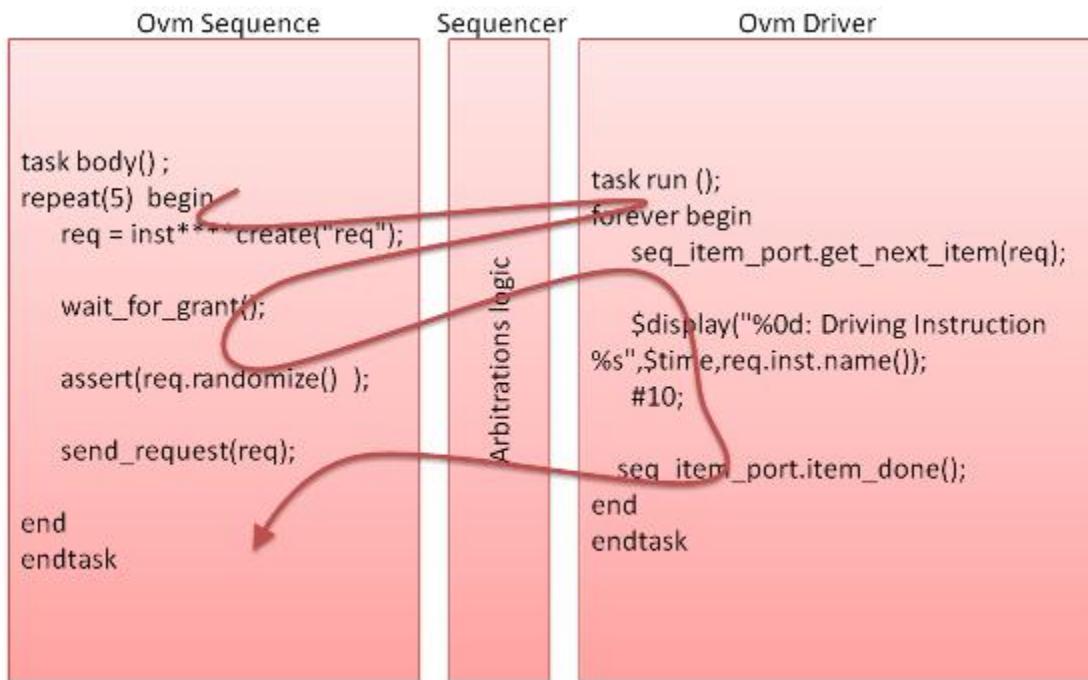


The above image shows how a transaction from a sequence is sent to driver and the response from the driver is sent to sequencer. There are multiple methods called during this operation.

First when the body() method is called

- 1) A transaction is created using "create()" method. If a transaction is created using "create()" method, then it can be overridden if required using ovm factory.
- 2) After a transaction is created, `wait_for_grant()` method is called. This method is blocking method.
- 3) In the run task of the driver, when "`seq_item_port.get_next_item()`" is called, then the sequencer un blocks `wait_for_grant()` method. If more than one sequence is getting executed by sequencer, then based on arbitration rules, un blocks the `wait_for_grant()` method.
- 4) After the `wait_for_grant()` un blocks, then transaction can be randomized, or its properties can be filled directly. Then using the `send_request()` method, send the transaction to the driver.
- 5) After calling the `send_request()` method, "`wait_for_item_done()`" method is called. This is a blocking method and execution gets blocks at this method call.
- 6) The transaction which is sent from sequence , in the driver this transaction is available as "`seq_item_port.get_next_item(req)`" method argument. Then driver can drive this transaction to bus or lower level.
- 7) Once the driver operations are completed, then by calling "`seq_item_port.put(rsp)`", `wait_for_item_done()` method of sequence gets unblocked. Using `get_responce(res)`, the response transaction from driver is taken by sequence and processes it.

After this step, again the steps 1 to 7 are repeated five times.



If a response from driver is not required, then steps 5,6,7 can be skipped and item\_done() method from driver should be called as shown in above image.

### Simple Example

Let write an example: This is a simple example of processor instruction. Various instructions which are supported by the processor are PUSH\_A,PUSH\_B,ADD,SUB,MUL,DIV and POP\_C.

### Sequence Item

1) Extend ovm\_sequence\_item and define instruction class.

```
class instruction extends ovm_sequence_item;
```

2) Define the instruction as enumerated types and declare a variable of instruction enumerated type.

```
typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
rand inst_t inst;
```

3) Define operational method using ovm\_field\_\* macros.

```
`ovm_object_utils_begin(instruction)
`ovm_field_enum(inst_t,inst,OVM_ALL_ON)
`ovm_object_utils_end
```

4) Define the constructor.

```
function new (string name = "instruction");
    super.new(name);
endfunction
```

### Sequence item code:

```
class instruction extends ovm_sequence_item;
  typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
  rand inst_t inst;

`ovm_object_utils_begin(instruction)
  `ovm_field_enum(inst_t,inst, OVM_ALL_ON)
`ovm_object_utils_end

function new (string name = "instruction");
  super.new(name);
endfunction

endclass
```

### Sequence

We will define a operation addition using ovm\_sequence. The instruction sequence should be "PUSH A PUSH B ADD POP C".

1) Define a sequence by extending ovm\_sequence. Set REQ parameter to "instruction" type.

```
class operation_addition extends ovm_sequence #(instruction);
```

2) Define the constructor.

```
function new(string name="operation_addition");
  super.new(name);
endfunction
```

3) Lets name the sequencer which we will develop is "instruction\_sequencer".

Using the `ovm\_sequence\_utils macro, register the "operation\_addition" sequence with "instruction\_sequencer" sequencer. This macro adds the sequence to the sequencer list. This macro will also register the sequence for factory overrides.

```
`ovm_sequence_utils(operation_addition, instruction_sequencer)
```

4)

In the body() method, first call wait\_for\_grant(), then construct a transaction and set the instruction enum to PUSH\_A . Then send the transaction to driver using send\_request() method. Then call the wait\_for\_item\_done() method. Repeat the above steps for other instructions PUSH\_B, ADD and POP\_C.

For construction of a transaction, we will use the create() method.

```
virtual task body();
  req = instruction::type_id::create("req");
  wait_for_grant();
```

```

assert(req.randomize() with {
    inst == instruction::PUSH_A;
});
send_request(req);
wait_for_item_done();
//get_response(res); This is optional. Not using in this example.

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::PUSH_B;
send_request(req);
wait_for_item_done();
//get_response(res);

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::ADD;
send_request(req);
wait_for_item_done();
//get_response(res);

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::POP_C;
send_request(req);
wait_for_item_done();
//get_response(res);

endtask

```

#### Sequence code

```

class operation_addition extends ovm_sequence #(instruction);
instruction req;
function new(string name="operation_addition");
    super.new(name);
endfunction
`ovm_sequence_utils(operation_addition, instruction_sequencer)
virtual task body();
    req = instruction::type_id::create("req");
    wait_for_grant();
    assert(req.randomize() with {
        inst == instruction::PUSH_A;
    });

```

```

    });

send_request(req);
wait_for_item_done();
//get_response(res); This is optional. Not using in this example.

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::PUSH_B;
send_request(req);
wait_for_item_done();
//get_response(res);

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::ADD;
send_request(req);
wait_for_item_done();
//get_response(res);

req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::POP_C;
send_request(req);
wait_for_item_done();
//get_response(res);
endtask

```

**endclass**

### Sequencer:

Ovm\_sequence has a property called default\_sequence. Default sequence is a sequence which will be started automatically. Using set\_config\_string, user can override the default sequence to any user defined sequence, so that when a sequencer is started, automatically a user defined sequence will be started. If over rides are not done with user defined sequence, then a random transaction are generated. Using "start\_default\_sequence()" method, "default\_sequence" can also be started.

Ovm sequencer has seq\_item\_export and res\_export tlm ports for connecting to ovm driver.

- 1) Define instruction\_sequencer by extending ovm\_sequencer.

```
class instruction_sequencer extends ovm_sequencer #(instruction);
```

2) Define the constructor.

Inside the constructor, place the macro `ovm\_update\_sequence\_lib\_and\_item().

This macro creates 3 predefined sequences. We will discuss about the predefined sequences in next section.

```
function new (string name, ovm_component parent);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(instruction)
endfunction
```

3) Place the ovm\_sequencer\_utils macro. This macro registers the sequencer for factory overrides.

```
`ovm_sequencer_utils(instruction_sequencer)
```

#### Sequencer Code:

```
class instruction_sequencer extends ovm_sequencer #(instruction);
```

```
function new (string name, ovm_component parent);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(instruction)
endfunction
```

```
`ovm_sequencer_utils(instruction_sequencer)
```

```
endclass
```

#### Driver:

ovm\_driver is a class which is extended from ovm\_componenet. This driver is used in pull mode. Pull mode means, driver pulls the transaction from the sequencer when it requires. Ovm driver has 2 TLM ports.

- 1) Seq\_item\_port: To get a item from sequencer, driver uses this port. Driver can also send response back using this port.
- 2) Rsp\_port : This can also be used to send response back to sequencer.

#### Seq\_item\_port methods:

Method	Type	Description
Task get_next_item(output REQreg_arg);	Blocking	Retrieves the next available item from a sequence.
task try_next_item( output REQreq_arg )	Non blocking	Retrieves the next available item from a sequence if one is available.
void item_done( RSP rsp_arg = null )	Non Blocking	Indicates that the request is completed to the sequencer.
task get( output REQreq_arg )	Blocking	Retrieves the next available item from a sequence.
task put( RSP rsp_arg )	Non Blocking	Sends a response back to the sequence that issued the request.
task peek( output REQreq_arg )	Blocking	Returns the current request item if one is in the sequencer fifo.

Lets implement a driver:

1) Define a driver which takes the instruction from the sequencer and does the processing. In this example we will just print the instruction type and wait for some delay.

```
class instruction_driver extends ovm_driver #(instruction);
```

2) Place the ovm\_component\_utils macro to define virtual methods like get\_type\_name and create.

```
`ovm_component_utils(instruction_driver)
```

3) Define Constructor method.

```
function new (string name, ovm_component parent);
    super.new(name, parent);
endfunction
```

4) Define the run() method. Run() method is executed in the "run phase". In this methods, transactions are taken from the sequencer and drive them on to dut interface or to other components.

Driver class has a port "seq\_item\_port". Using the method seq\_item\_port.get\_next\_item(), get the transaction from the sequencer and process it. Once the processing is done, using the item\_done() method, indicate to the sequencer that the request is completed. In this example, after taking the transaction, we will print the transaction and wait for 10 units time.

```
task run ();
    while(1) begin
        seq_item_port.get_next_item(req);
        $display("%0d: Driving Instruction %s",$time,req.inst.name());
        #10;
        seq_item_port.item_done();
```

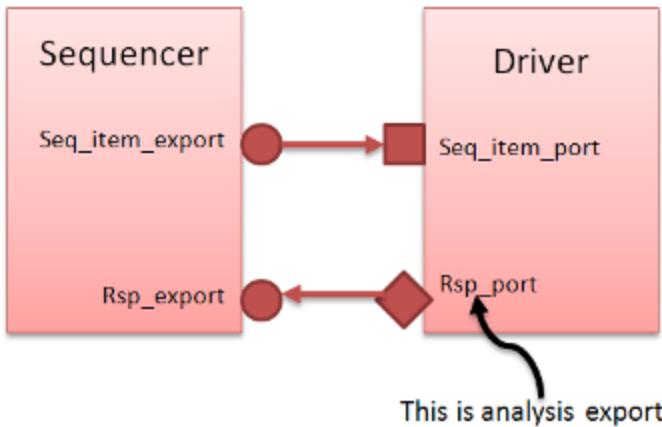
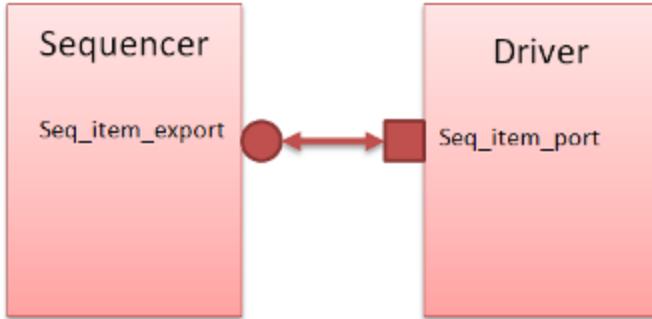
```
end  
endtask  
endclass
```

#### Driver class code:

```
class instruction_driver extends ovm_driver #(instruction);  
  // Provide implementations of virtual methods such as get_type_name and create  
  `ovm_component_utils(instruction_driver)  
  // Constructor  
  function new (string name, ovm_component parent);  
    super.new(name, parent);  
  endfunction  
  task run ();  
    forever begin  
      seq_item_port.get_next_item(req);  
      $display("%0d: Driving Instruction %s",$time,req.inst.name());  
      #10;  
      // rsp.set_id_info(req); These two steps are required only if  
      // seq_item_port.put(esp); response needs to be sent back to sequence  
      seq_item_port.item_done();  
    end  
  endtask  
endclass
```

#### Driver And Sequencer Connectivity:

Deriver and sequencer are connected using TLM. Ovm\_driver has seq\_item\_port which is used to get the transaction from ovm sequencer. This port is connected to ovm\_sequencer seq\_item\_export Using "<driver>.seq\_item\_port.connect(<sequencer>.seq\_item\_export);" driver and sequencer can be connected. Simillarly "res\_port" of driver which is used to send response from driver to sequencer is connected to "res\_export" of the sequencer using ""<driver>.res\_port.connect(<sequencer>.res\_export);".



### Testcase:

This testcase is used only for the demo purpose of this tutorial session. Actually, the sequencer and the driver are instantiated and their ports are connected in a agent component and used. Lets implement a testcase

1) Take instances of sequencer and driver and construct both components.

```

sequencer = new("sequencer", null);
sequencer.build();
driver = new("driver", null);
driver.build();
  
```

2)

Connect the seq\_item\_export to the drivers seq\_item\_port.

```
driver.seq_item_port.connect(sequencer.seq_item_export);
```

3) Using set\_config\_string() method, set the default sequence of the sequencer to "operation\_addition". Operation\_addition is the sequence which we defined previous.

```
set_config_string("sequencer", "default_sequence", "operation_addition");
```

4) Using the start\_default\_sequence() method of the sequencer, start the default sequence of the sequencer. In the previous step we configured the addition operation as default sequence. When

you run the simulation, you will see the PUSH\_A,PUSH\_B ADD and POP\_C series of transaction.

```
sequencer.start_default_sequence();
```

**Testcase Code:**

```
module test;
instruction_sequencer sequencer;
instruction_driver driver;
initial begin
    set_config_string("sequencer", "default_sequence", "operation_addition");
    sequencer = new("sequencer", null);
    sequencer.build();
    driver = new("driver", null);
    driver.build();

    driver.seq_item_port.connect(sequencer.seq_item_export);
    sequencer.print();
    fork
        begin
            run_test();
            sequencer.start_default_sequence();
        end
        #2000 global_stop_request();
    join
end
endmodule
```

**Download the example:**

[ovm\\_basic\\_sequence.tar](#)

[Browse the code in ovm\\_basic\\_sequence.tar](#)

**Command to simulate**

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

**Log file Output**

OVM\_INFO @ 0 [RNTST] Running test ...

0: Driving Instruction PUSH\_A

10: Driving Instruction PUSH\_B

20: Driving Instruction ADD

30: Driving Instruction POP\_C

From the above log , we can see that transactions are generates as we defined in ovm sequence.

## **OVM SEQUENCE 2**

### **Pre Defined Sequences:**

Every sequencer in ovm has 3 pre defined sequences. They are

- ② 1)Ovm\_random\_sequence
- ② 2)Ovm\_exhaustive\_sequence.
- ② 3)Ovm\_simple\_sequence

All the user defined sequences which are registered by user and the above three predefined sequences are stored in sequencer queue.

<b>id</b>	<b>Sequences[\$]</b>
0	Ovm_random_sequence
1	Ovm_exhaustive_sequence
2	Ovm_simple_sequence
3	User defined sequence 1
4	User defined sequence 2
5	User defined sequence 3
.	...
.	...

© [www.testbench.in](http://www.testbench.in)

### **Ovm\_random\_sequence :**

This sequence randomly selects and executes a sequence from the sequencer sequence library, excluding ovm\_random\_sequence itself, and ovm\_exhaustive\_sequence. From the above image, from sequence id 2 to till the last sequence, all the sequences are executed randomly. If the "count" variable of the sequencer is set to 0, then none of the sequence is executed. If the "count" variable of the sequencer is set to -1, then some random number of sequences from 0 to "max\_random\_count" are executed. By default "mac\_random\_count" is set to 10. "Count" and "mac\_random\_count" can be changed using set\_config\_int().

The sequencer when automatically started executes the sequence which is point by default\_sequence. By default default\_sequence variable points to ovm\_random\_sequence.

### **ovm\_exhaustive\_sequence:**

This sequence randomly selects and executes each sequence from the sequencers sequence library once in a randc style, excluding itself and ovm\_random\_sequence.

### **ovm\_simple\_sequence:**

This sequence simply executes a single sequence item.

In the previous example from OVM\_SEQUENCE\_1 section.  
The print() method of the sequencer in that example printed the following

Name	Type	Size	Value
sequencer	instruction_sequen+ -		sequencer@2
rsp_export	ovm_analysis_export -		rsp_export@4
seq_item_export	ovm_seq_item_pull_+ -		seq_item_export@28
default_sequence	string	18	operation_addition
count	integral	32	-1
max_random_count	integral	32	'd10
sequences	array	4	-
[0]	string	19	ovm_random_sequence
[1]	string	23	ovm_exhaustive_sequ+
[2]	string	19	ovm_simple_sequence
[3]	string	18	operation_addition
max_random_depth	integral	32	'd4
num_last_reqs	integral	32	'd1
num_last_rsps	integral	32	'd1

Some observations from the above log:

The count is set to -1. The default sequencer is set to operations\_addition. There are 3 predefined sequences and 1 user defined sequence.

Lets look at a example: In the attached example, in file sequence.sv file, there are 4 sequences, they are operation\_addition, operation\_subtraction, operation\_multiplication.

In the testcase.sv file, the "default\_sequence" is set to "ovm\_exhaustive\_sequence" using the set\_config\_string.

```
set_config_string("sequencer", "default_sequence", "ovm_exhaustive_sequence");
```

### Download the example

[ovm\\_sequence\\_1.tar](#)

[Browse the code in ovm\\_sequence\\_1.tar](#)

### Command to run the summation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

### Log File

0: Driving Instruction PUSH\_B

10: Driving Instruction PUSH\_A  
20: Driving Instruction PUSH\_B  
30: Driving Instruction SUB  
40: Driving Instruction POP\_C

50: Driving Instruction PUSH\_A  
60: Driving Instruction PUSH\_B  
70: Driving Instruction MUL  
80: Driving Instruction POP\_C

90: Driving Instruction PUSH\_A  
100: Driving Instruction PUSH\_B  
110: Driving Instruction ADD  
120: Driving Instruction POP\_C

From the above log , we can see that all the 3 user defined sequences and predefined ovm\_simple\_sequence are executed.

#### **Sequence Action Macro:**

In the previous sections, we have seen the implementation of body() method of sequence. The body() method implementation requires some steps. We have seen these steps as Creation of item, wait for grant, randomize the item, send the item.

All these steps have been automated using "sequence action macros". There are some more additional steps added in these macros. Following are the steps defined with the "sequence action macro".

	<b>Phase</b>	<b>Operation</b>
1	Create	Construct a transaction using create() method.
2	Synchronize	Call wait_for_grant() method
3	Pre_do	Call pre_do() method.
4	Randomize	Optionally Randomize transaction.
5	Mid_do	Call mid_do() method.
6	Post_Synchronize	Call send_request() and wait_for_item_done() methods.
7	Post_do	Optionally call post_do() and get_response() methods.

Pre\_do(), mid\_do() and post\_do() are callback methods which are in ovm sequence. If user is

interested , he can use these methods. For example, in mid\_do() method, user can print the transaction or the randomized transaction can be fined tuned. These methods should not be called by user directly.

**Syntax:**

```
virtual task pre_do(bit is_item)  
virtual function void mid_do(ovm_sequence_item this_item)  
virtual function void post_do(ovm_sequence_item this_item)
```

Pre\_do() is a task , if the method consumes simulation cycles, the behavior may be unexpected.

**Example Of Pre do,Mid do And Post do**

Lets look at a example: We will define a sequence using `ovm\_do macro. This macro has all the above defined phases.

1)Define the body method using the `ovm\_do() macro. Before and after this macro, just call messages.

```
virtual task body();  
    ovm_report_info(get_full_name(),  
        "Seuqnce Action Macro Phase : Before ovm_do macro ",OVM_LOW);  
    `ovm_do(req);  
    ovm_report_info(get_full_name(),  
        "Seuqnce Action Macro Phase : After ovm_do macro ",OVM_LOW);  
endtask
```

2)Define pre\_do() method. Lets just print a message from this method.

```
virtual task pre_do(bit is_item);  
    ovm_report_info(get_full_name(),  
        "Seuqnce Action Macro Phase : PRE_DO ",OVM_LOW);  
endtask
```

3)Define mid\_do() method. Lets just print a message from this method.

```
virtual function void mid_do(ovm_sequence_item this_item);  
    ovm_report_info(get_full_name(),  
        "Seuqnce Action Macro Phase : MID_DO ",OVM_LOW);  
endfunction
```

4)Define post\_do() method. Lets just print a message from this method.

```
virtual function void post_do(ovm_sequence_item this_item);  
    ovm_report_info(get_full_name(),  
        "Seuqnce Action Macro Phase : POST_DO ",OVM_LOW);  
endfunction
```

### Complet sequence code:

```
class demo_ovm_do extends ovm_sequence #(instruction);
  instruction req;
  function new(string name="demo_ovm_do");
    super.new(name);
  endfunction
  `ovm_sequence_utils(demo_ovm_do, instruction_sequencer)
  virtual task pre_do(bit is_item);
    ovm_report_info(get_full_name(),
      "Seuqnce Action Macro Phase : PRE_DO ",OVM_LOW);
  endtask
  virtual function void mid_do(ovm_sequence_item this_item);
    ovm_report_info(get_full_name(),
      "Seuqnce Action Macro Phase : MID_DO ",OVM_LOW);
  endfunction
  virtual function void post_do(ovm_sequence_item this_item);
    ovm_report_info(get_full_name(),
      "Seuqnce Action Macro Phase : POST_DO ",OVM_LOW);
  endfunction
  virtual task body();
    ovm_report_info(get_full_name(),
      "Seuqnce Action Macro Phase : Before ovm_do macro ",OVM_LOW);
    `ovm_do(req);
    ovm_report_info(get_full_name(),
      "Seuqnce Action Macro Phase : After ovm_do macro ",OVM_LOW);
  endtask
endclass
```

### Download the example

[ovm\\_sequence\\_2.tar](#)

[Browse the code in ovm\\_sequence\\_2.tar](#)

### Command to run the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

### Log file report:

```
OVM_INFO@0:reporter[sequencer.demo_ovm_do]
  Seuqnce Action Macro Phase : Before ovm_do macro
OVM_INFO@0:reporter[sequencer.demo_ovm_do]
  Seuqnce Action Macro Phase : PRE_DO
OVM_INFO@0:reporter[sequencer.demo_ovm_do]
  Seuqnce Action Macro Phase : MID_DO
```

0: Driving Instruction MUL

OVM\_INFO@10:reporter[sequencer.demo\_ovm\_do]

Seuqnce Action Macro Phase : POST\_DO

OVM\_INFO@10:reporter[sequencer.demo\_ovm\_do]

Seuqnce Action Macro Phase : After ovm\_do macro

The above log file shows the messages from pre\_do,mid\_do and post\_do methods.

### **List Of Sequence Action Macros:**

These macros are used to start sequences and sequence items that were either registered with a <ovm-sequence\_utils> macro or whose associated sequencer was already set using the <set\_sequencer> method.

#### **`ovm\_create(item/sequence)**

This action creates the item or sequence using the factory. Only the create phase will be executed.

#### **`ovm do(item/sequence)**

This macro takes as an argument a ovm\_sequence\_item variable or sequence . All the above defined 7 phases will be executed.

#### **`ovm do with(item/sequence, Constraint block)**

This is the same as `ovm\_do except that the constraint block in the 2nd argument is applied to the item or sequence in a randomize with statement before execution.

#### **`ovm send(item/sequence)**

Create phase and randomize phases are skipped, rest all the phases will be executed. Using `ovm\_create, create phase can be executed. Essentially, an `ovm\_do without the create or randomization.

#### **`ovm rand send(item/sequence)**

Only create phase is skipped. rest of all the phases will be executed. User should use `ovm\_create to create the sequence or item.

#### **`ovm rand send with(item/sequence , Constraint block)**

Only create phase is skipped. rest of all the phases will be executed. User should use `ovm\_create to create the sequence or item. Constraint block will be applied which randomization.

#### **`ovm do pri(item/sequence, priority )**

This is the same as `ovm\_do except that the sequence item or sequence is executed with the priority specified in the argument.

#### **`ovm do pri with(item/sequence , constraint block , priority)**

This is the same as `ovm\_do\_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

#### **`ovm send pri(item/sequence,priority)**

This is the same as `ovm\_send except that the sequence item or sequence is executed with the priority specified in the argument.

### `ovm\_rand\_send\_pri(item/sequence,priority)

This is the same as `ovm\_rand\_send except that the sequence item or sequence is executed with the priority specified in the argument.

### `ovm\_rand\_send\_pri\_with(item/sequence,priority,constraint block)

This is the same as `ovm\_rand\_send\_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

Following macros are used on sequence or sequence items on a different sequencer.

### `ovm\_create\_on(item/sequence,sequencer)

This is the same as `ovm\_create except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

### `ovm\_do\_on(item/sequence,sequencer)

This is the same as `ovm\_do except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

### `ovm\_do\_on\_pri(item/sequence,sequencer, priority)

This is the same as `ovm\_do\_pri except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

### `ovm\_do\_on\_with(item/sequence,sequencer, constraint block)

This is the same as `ovm\_do\_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument. The user must supply brackets around the constraints.

### `ovm\_do\_on\_pri\_with(item/sequence,sequencer,priority,constraint block)

This is the same as `ovm\_do\_pri\_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified sequencer argument.

## Examples With Sequence Action Macros:

### **virtual task** body();

```
ovm_report_info(get_full_name(),
    "Executing Sequence Action Macro ovm_do",OVM_LOW);  
`ovm_do(req)
```

**endtask**

### **virtual task** body();

```
ovm_report_info(get_full_name(),
    "Executing Sequence Action Macro ovm_do_with ",OVM_LOW);  
`ovm_do_with(req,{ inst == ADD; })
```

**endtask**

### **virtual task** body();

```
ovm_report_info(get_full_name(),
    "Executing Sequence Action Macro ovm_create and ovm_send",OVM_LOW);  
`ovm_create(req)  
req.inst = instruction::PUSH_B;
```

```

`ovm_send(req)
endtask
virtual task body();
    ovm_report_info(get_full_name(),
        "Executing Sequence Action Macro ovm_create and ovm_rand_send",OVM_LOW);
    `ovm_create(req)
    `ovm_rand_send(req)
endtask

```

### [Download the example](#)

[ovm\\_sequence\\_3.tar](#)

[Browse the code in ovm\\_sequence\\_3.tar](#)

### Command to run the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

### Log file report

```

0: Driving Instruction PUSH_B
OVM_INFO@10:reporter[***]Executing Sequence Action Macro ovm_do_with
10: Driving Instruction ADD
OVM_INFO@20:reporter[***]Executing Sequence Action Macro ovm_create and ovm_send
20: Driving Instruction PUSH_B
OVM_INFO@30:reporter[***]Executing Sequence Action Macro ovm_do
30: Driving Instruction DIV
OVM_INFO@40:reporter[***]Executing Sequence Action Macro ovm_create and
ovm_rand_send
40: Driving Instruction MUL

```

## **OVM SEQUENCE 3**

### Body Callbacks:

Ovm sequences has two callback methods pre\_body() and post\_body(), which are executed before and after the sequence body() method execution. These callbacks are called only when start\_sequence() of sequencer or start() method of the sequence is called. User should not call these methods.

**virtual task** pre\_body()

**virtual task** post\_body()

### Example

In this example, I just printed messages from pre\_body() and post\_body() methods. These methods can be used for initialization, synchronization with some events or cleanup.

```

class demo_pre_body_post_body extends ovm_sequence #(instruction);
    instruction req;
    function new(string name="demo_pre_body_post_body");
        super.new(name);
    endfunction

```

```

`ovm_sequence_utils(demo_pre_body_post_body, instruction_sequencer)
virtual task pre_body();
    ovm_report_info(get_full_name()," pre_body() callback ",OVM_LOW);
endtask
virtual task post_body();
    ovm_report_info(get_full_name()," post_body() callback ",OVM_LOW);
endtask
virtual task body();
    ovm_report_info(get_full_name(),
        "body() method: Before ovm_do macro ",OVM_LOW);
    `ovm_do(req);
    ovm_report_info(get_full_name(),
        "body() method: After ovm_do macro ",OVM_LOW);
endtask
endclass

```

### Download the example

[ovm\\_sequence\\_4.tar](#)

[Browse the code in ovm\\_sequence\\_4.tar](#)

### Command to sun the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

### Log file report

```

OVM_INFO @ 0 [RNTST] Running test ...
OVM_INFO @ 0: reporter [***] pre_body() callback
OVM_INFO @ 0: reporter [***] body() method: Before ovm_do macro
0: Driving Instruction SUB
OVM_INFO @ 10: reporter [***] body() method: After ovm_do macro
OVM_INFO @ 10: reporter [***] post_body() callback

```

### Hierarchical Sequences

One main advantage of sequences is smaller sequences can be used to create sequences to generate stimulus required for todays complex protocol.

To create a sequence using another sequence, following steps has to be done

- 1)Extend the ovm\_sequence class and define a new class.
- 2)Declare instances of child sequences which will be used to create new sequence.
- 3)Start the child sequence using <instance>.start() method in body() method.

### Sequential Sequences

To executes child sequences sequentially, child sequence start() method should be called sequentially in body method.

In the below example you can see all the 3 steps mentioned above.

In this example, I have defined 2 child sequences. These child sequences can be used as normal sequences.

### Sequence 1 code:

This sequence generates 4 PUSH\_A instructions.

```
virtual task body();
  repeat(4) begin
    `ovm_do_with(req, { inst == PUSH_A; });
  end
endtask
```

### Sequence 2 code:

This sequence generates 4 PUSH\_B instructions.

```
virtual task body();
  repeat(4) begin
    `ovm_do_with(req, { inst == PUSH_B; });
  end
endtask
```

### Sequential Sequence code:

This sequence first calls sequence 1 and then calls sequence 2.

```
class sequential_sequence extends ovm_sequence #(instruction);
seq_a s_a;
seq_b s_b;
function new(string name="sequential_sequence");
  super.new(name);
endfunction
`ovm_sequence_utils(sequential_sequence, instruction_sequencer)
virtual task body();
  `ovm_do(s_a);
  `ovm_do(s_b);
endtask
endclass
```

From the testcase, "sequential\_sequence" is selected as "default\_sequence".

### Download the example

[ovm\\_sequence\\_5.tar](#)

[Browse the code in ovm\\_sequence\\_5.tar](#)

### Command to run the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

### Log file report

```
0: Driving Instruction PUSH_A
10: Driving Instruction PUSH_A
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_A
40: Driving Instruction PUSH_B
```

```
50: Driving Instruction PUSH_B  
60: Driving Instruction PUSH_B  
70: Driving Instruction PUSH_B
```

If you observe the above log, you can see sequence seq\_a is executed first and then sequence seq\_b is executed.

### Parallel sequences

To execute child sequences parallel, child sequence start() method should be called parallel using fork/join in body method.

#### Parallel Sequence code:

```
class parallel_sequence extends ovm_sequence #(instruction);  
    seq_a s_a;  
    seq_b s_b;  
    function new(string name="parallel_sequence");  
        super.new(name);  
    endfunction  
    `ovm_sequence_utils(parallel_sequence, instruction_sequencer)  
    virtual task body();  
        fork  
            `ovm_do(s_a)  
            `ovm_do(s_b)  
        join  
    endtask  
endclass
```

#### Download the example

[ovm\\_sequence\\_6.tar](#)

[Browse the code in ovm\\_sequence\\_6.tar](#)

#### Command to run the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

#### Log file report

```
OVM_INFO @ 0 [RNTST] Running test ...  
0: Driving Instruction PUSH_A  
10: Driving Instruction PUSH_B  
20: Driving Instruction PUSH_A  
30: Driving Instruction PUSH_B  
40: Driving Instruction PUSH_A  
50: Driving Instruction PUSH_B  
60: Driving Instruction PUSH_A  
70: Driving Instruction PUSH_B
```

## OVM SEQUENCE 4

### Sequencer Arbitration:

When sequencers are executed parallel, sequencer will arbitrate among the parallel sequence. When all the parallel sequences are waiting for a grant from sequencer using `wait_for_grant()` method, then the sequencer, using the arbitration mechanism, sequencer grants to one of the sequencer.

There are 6 different arbitration algorithms, they are

Macro	Algorithm
SEQ_ARB_FIFO	Requests are granted in FIFO style. Priorities will not be considered.
SEQ_ARB_RANDOM	Requests are granted randomly. Priorities will not be considered.
SEQ_ARB_WEIGHTED	Based on the priority value, randomly grants the requests.
SEQ_ARB_STRICT_FIFO	Sequences with high priority value will be granted in FIFO style.
SEQ_ARB_STRICT_RANDOM	Sequences with high priority value will be granted randomly.
SEQ_ARB_USER	Grants are done based on user defined algorithm.

© www.testbench.in

To set the arbitration, use the `set_arbitration()` method of the sequencer. By default , the arbitration algorithms is set to SEQ\_ARB\_FIFO.

**function void** set\_arbitration(**SEQ\_ARB\_TYPE** val)

Lets look at a example.

In this example, I have 3 child sequences seq\_mul seq\_add and seq\_sub each of them generates 3 transactions.

### Sequence code 1:

```
virtual task body();
    repeat(3) begin
        `ovm_do_with(req, { inst == MUL; });
    end
endtask
```

### Sequence code 2:

```
virtual task body();
    repeat(3) begin
        `ovm_do_with(req, { inst == ADD; });
    end
endtask
```

### Sequence code 3:

```
virtual task body();
    repeat(3) begin
        `ovm_do_with(req, { inst == SUB; });
    end
endtask
```

### Parallel sequence code:

In the body method, before starting child sequences, set the arbitration using set\_arbitration(). In this code, im setting it to SEQ\_ARB\_RANDOM.

```
class parallel_sequence extends ovm_sequence #(instruction);
```

```
seq_add add;
seq_sub sub;
seq_mul mul;
```

```
function new(string name="parallel_sequence");
    super.new(name);
endfunction
```

```
`ovm_sequence_utils(parallel_sequence, instruction_sequencer)
```

```
virtual task body();
    m_sequencer.set_arbitration(SEQ_ARB_RANDOM);
    fork
        `ovm_do(add)
        `ovm_do(sub)
        `ovm_do(mul)
    join
endtask
```

```
endclass
```

### Download the example

[ovm\\_sequence\\_7.tar](#)

[Browse the code in ovm\\_sequence\\_7.tar](#)

### Command to sun the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

### Log file report for when SEQ\_ARB\_RANDOM is set.

```
0: Driving Instruction MUL  
10: Driving Instruction SUB  
20: Driving Instruction MUL  
30: Driving Instruction SUB  
40: Driving Instruction MUL  
50: Driving Instruction ADD  
60: Driving Instruction ADD  
70: Driving Instruction SUB  
80: Driving Instruction ADD
```

### Log file report for when SEQ\_ARB\_FIFO is set.

```
0: Driving Instruction ADD  
10: Driving Instruction SUB  
20: Driving Instruction MUL  
30: Driving Instruction ADD  
40: Driving Instruction SUB  
50: Driving Instruction MUL  
60: Driving Instruction ADD  
70: Driving Instruction SUB  
80: Driving Instruction MUL
```

If you observe the first log report, all the transaction of the sequences are generated in random order. In the second log file, the transactions are given equal priority and are in fifo order.

#### Setting The Sequence Priority:

There are two ways to set the priority of a sequence. One is using the start method of the sequence and other using the set\_priority() method of the sequence. By default, the priority of a sequence is 100. Higher numbers indicate higher priority.

```
virtual task start (ovm_sequencer_base sequencer,  
                      ovm_sequence_base parent_sequence = null,  
                      integer this_priority = 100,  
                      bit call_pre_post = 1)
```

```
function void set_priority (int value)
```

Lets look a example with SEQ\_ARB\_WEIGHTED.

For sequence seq\_mul set the weight to 200.

For sequence seq\_add set the weight to 300.

For sequence seq\_sub set the weight to 400.

In the below example, start() method is used to override the default priority value.

Code :

```
class parallel_sequence extends ovm_sequence #(instruction);
    seq_add add;
    seq_sub sub;
    seq_mul mul;
    function new(string name="parallel_sequence");
        super.new(name);
    endfunction
    `ovm_sequence_utils(parallel_sequence, instruction_sequencer)
    virtual task body();
        m_sequencer.set_arbitration(SEQ_ARB_WEIGHTED);
        add = new("add");
        sub = new("sub");
        mul = new("mul");
        fork
            sub.start(m_sequencer,this,400);
            add.start(m_sequencer,this,300);
            mul.start(m_sequencer,this,200);
        join
    endtask
endclass
```

Download the example

[ovm\\_sequence\\_8.tar](#)

Browse the code in ovm\_sequence\_8.tar

Command to run the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

Log file report

```
0: Driving Instruction MUL
10: Driving Instruction ADD
20: Driving Instruction SUB
30: Driving Instruction SUB
40: Driving Instruction ADD
50: Driving Instruction ADD
60: Driving Instruction ADD
70: Driving Instruction MUL
80: Driving Instruction SUB
```

## OVM SEQUENCE 5

### Sequencer Registration Macros

Sequence Registration Macros does the following

- 1) Implements get\_type\_name method.
- 2) Implements create() method.
- 3) Registers with the factory.
- 4) Implements the static get\_type() method.
- 5) Implements the virtual get\_object\_type() method.
- 6) Registers the sequence type with the sequencer type.
- 7) Defines p\_sequencer variable. p\_sequencer is a handle to its sequencer.
- 8) Implements m\_set\_p\_sequencer() method.

If there are no local variables, then use following macro

```
`ovm_sequence_utils(TYPE_NAME,SQR_TYPE_NAME)
```

If there are local variables in sequence, then use macro

```
`ovm_sequence_utils_begin(TYPE_NAME,SQR_TYPE_NAME)
```

```
 `ovm_field_* macro invocations here
```

```
`ovm_sequence_utils_end
```

Macros `ovm\_field\_\* are used for define utility methods.

These `ovm\_field\_\* macros are discussed in

### OVM TRANSACTION

Example to demonstrate the usage of the above macros:

```
class seq_mul extends ovm_sequence #(instruction);
  rand integer num_inst ;

  instruction req;
  constraint num_c { num_inst inside { 3,5,7 }; };

  `ovm_sequence_utils_begin(seq_mul,instruction_sequencer)
    `ovm_field_int(num_inst, OVM_ALL_ON)
  `ovm_sequence_utils_end

  function new(string name="seq_mul");
    super.new(name);
  endfunction

  virtual task body();
    ovm_report_info(get_full_name(),
      $psprintf("Num of transactions %d",num_inst),OVM_LOW);
    repeat(num_inst) begin
      `ovm_do_with(req, { inst == MUL; });
    end
  endtask

endclass
```

## Download the example

[ovm\\_sequence\\_9.tar](#)

[Browse the code in ovm\\_sequence\\_9.tar](#)

## Command to run the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

## Log

OVM\_INFO @ 0: reporter [RNTST] Running test ...  
OVM\_INFO @ 0: reporter [sequencer.seq\_mul] Num of transactions 5  
0: Driving Instruction MUL  
10: Driving Instruction MUL  
20: Driving Instruction MUL  
30: Driving Instruction MUL  
40: Driving Instruction MUL

## Setting Sequence Members:

set\_config\_\* can be used only for the components not for the sequences. By using configuration you can change the variables inside components only not in sequences.

But there is a workaround to this problem.

Sequence has handle name called p\_sequencer which is pointing the Sequencer on which it is running.

Sequencer is a component , so get\_config\_\* methods are implemented for it.

So from the sequence, using the sequencer get\_config\_\* methods, sequence members can be updated if the variable is configured.

When using set\_config\_\* , path to the variable should be sequencer name, as we are using the sequencer get\_config\_\* method.

Following method demonstrates how this can be done:

## Sequence:

- 1) num\_inst is a integer variables which can be updated.
- 2) In the body method, call the get\_config\_int() method to get the integer value if num\_inst is configured from testcase.

```
class seq_mul extends ovm_sequence #(instruction);  
    integer num_inst = 4;  
    instruction req;  
    `ovm_sequence_utils_begin(seq_mul,instruction_sequencer)  
    `ovm_field_int(num_inst, OVM_ALL_ON)  
    `ovm_sequence_utils_end  
    function new(string name="seq_mul");  
        super.new(name);  
    endfunction  
    virtual task body();
```

```

void'(p_sequencer.get_config_int("num_inst",num_inst));
ovm_report_info(get_full_name(),
    $psprintf("Num of transactions %d",num_inst),OVM_LOW);
repeat(num_inst) begin
    `ovm_do_with(req, { inst == MUL; });
end
endtask
endclass

```

**Testcase:**

From the testcase, using the set\_config\_int() method, configure the num\_inst to 3. The instance path argument should be the sequencer path name.

```

module test;
instruction_sequencer sequencer;
instruction_driver driver;
initial begin
    set_config_string("sequencer", "default_sequence", "seq_mul");
    set_config_int("sequencer", "num_inst",3);
    sequencer = new("sequencer", null);
    sequencer.build();
    driver = new("driver", null);
    driver.build();
    driver.seq_item_port.connect(sequencer.seq_item_export);
    sequencer.print();
fork
    begin
        run_test();
        sequencer.start_default_sequence();
    end
    #3000 global_stop_request();
join
end
endmodule

```

[Download the example](#)

[ovm\\_sequence\\_10.tar](#)

[Browse the code in ovm\\_sequence\\_10.tar](#)

**Command to sun the simulation**

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

## Log

OVM\_INFO @ 0: reporter [RNTST] Running test ...

OVM\_INFO @ 0: reporter [sequencer.seq\_mul] Num of transactions 3

0: Driving Instruction MUL

10: Driving Instruction MUL

20: Driving Instruction MUL

From the above log we can see that seq\_mul.num\_inst value is 3.

## **OVM SEQUENCE 6**

### **Exclusive Access**

A sequence may need exclusive access to the driver which sequencer is arbitrating among multiple sequence. Some operations require that a series of transaction needs to be driven without any other transaction in between them. Then a exclusive access to the driver will allow to a sequence to complete its operation with out any other sequence operations in between them. There are 2 mechanisms to get exclusive access:

- ⌚ Lock-unlock
- ⌚ Grab-ungrab

### **Lock-Unlock**

**task** lock(*ovm\_sequencer\_base sequencer = Null*)

**function void** unlock(*ovm\_sequencer\_base sequencer = Null*)

Using lock() method , a sequence can requests for exclusive access. A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence. A lock() is blocking task and when access is granted, it will unblock.

Using unlock(), removes any locks or grabs obtained by this sequence on the specified sequencer.

If sequencer is null, the lock/unlock will be applied on the current default sequencer.

Lets see an example,

In this example there are 3 sequences with each sequence generating 4 transactions. All these 3 sequences will be called in parallel in another sequence.

### **Sequence 1 code:**

```
virtual task body();  
    repeat(4) begin  
        `ovm_do_with(req, { inst == PUSH_A; });  
    end  
endtask
```

### **Sequence 2 code:**

```
virtual task body();  
    repeat(4) begin  
        `ovm_do_with(req, { inst == POP_C; });  
    end endtask
```

### Sequence 3 code:

In this sequence , call the lock() method to get the exclusive access to driver.  
After completing all the transaction driving, then call the unclock() method.

```
virtual task body();
    lock();
    repeat(4) begin
        `ovm_do_with(req, { inst == PUSH_B; });
    end
    unlock();
endtask
```

### Parallel sequence code:

```
virtual task body();
    fork
        `ovm_do(s_a)
        `ovm_do(s_b)
        `ovm_do(s_c)
    join
endtask
```

### Download the example

[ovm\\_sequence\\_11.tar](#)

[Browse the code in ovm\\_sequence\\_11.tar](#)

### Command to sun the simulation

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

### Log file:

```
0: Driving Instruction PUSH_A
10: Driving Instruction POP_C
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction POP_C
80: Driving Instruction PUSH_A
90: Driving Instruction POP_C
100: Driving Instruction PUSH_A
110: Driving Instruction POP_C
```

From the above log file, we can observe that , when seq\_b sequence got the access, then transactions from seq\_a and seq\_c are not generated.

Lock() will be arbitrated before giving the access. To get the exclusive access without arbitration, grab() method should be used.

## Grab-Ungrab

```
task grab(ovm_sequencer_base sequencer = null)
function void ungrab(ovm_sequencer_base sequencer = null)
```

grab() method requests a lock on the specified sequencer. A grab() request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab() is granted when no other grabs or locks are blocking this sequence.

A grab() is blocking task and when access is granted, it will unblock.

Ungrab() method removes any locks or grabs obtained by this sequence on the specified sequencer.

If no argument is supplied, then current default sequencer is chosen.

Example:

```
virtual task body();
#25;
grab();
repeat(4) begin
`ovm_do_with(req, { inst == PUSH_B; });
end
ungrab();
endtask
```

[Download the example](#)

[ovm\\_sequence\\_12.tar](#)

[Browse the code in ovm\\_sequence\\_12.tar](#)

[Command to run the simulation](#)

Your\_tool\_simulation\_command +incdir+path\_to\_ovm testcase.sv

```
0: Driving Instruction PUSH_A
10: Driving Instruction POP_C
20: Driving Instruction PUSH_A
30: Driving Instruction PUSH_B
40: Driving Instruction PUSH_B
50: Driving Instruction PUSH_B
60: Driving Instruction PUSH_B
70: Driving Instruction POP_C
80: Driving Instruction PUSH_A
90: Driving Instruction POP_C
100: Driving Instruction PUSH_A
110: Driving Instruction POP_C
```

## OVM CONFIGURATION

Configuration is a mechanism in OVM that higher level components in a hierarchy can configure the lower level components variables. Using set\_config\_\* methods, user can configure integer, string and objects of lower level components. Without this mechanism, user should

access the lower level component using hierarchy paths, which restricts reusability. This mechanism can be used only with components. Sequences and transactions cannot be configured using this mechanism. When set\_config\_\* method is called, the data is stored w.r.t strings in a table. There is also a global configuration table.

Higher level component can set the configuration data in level component table. It is the responsibility of the lower level component to get the data from the component table and update the appropriate table.

#### **Set config \*** Methods:

Following are the method to configure integer , string and object of ovm\_object based class respectively.

```
function void set_config_int (string inst_name,  
                           string field_name,  
                           ovm_bitstream_t value)  
function void set_config_string (string inst_name,  
                                string field_name,  
                                string value)  
function void set_config_object (string inst_name,  
                                string field_name,  
                                ovm_object value, bit clone = 1)
```

#### **Arguments description:**

- ⌚ **string** inst\_name: Hierarchical string path.
- ⌚ **string** field\_name: Name of the field in the table.
- ⌚ **bitstream\_t** value: In set\_config\_int, a integral value that can be anything from 1 bit to 4096 bits.
- ⌚ **bit** clone : If this bit is set then object is cloned.

inst\_name and field\_name are strings of hierachal path. They can include wile card "\*" and "?" characters. These methods must be called in build phase of the component.

"\*" matches zero or more characters

"?" matches exactly one character

#### **Some examples:**

"\*" -All the lower level components.

"\*abc" -All the lower level components which ends with "abc".

Example: "xabc","xyabc","xyzabc" ....

"abc\*" -All the lower level components which starts with "abc".

Example: "abcx","abcxy","abcxyz" ....

"ab?" -All the lower level components which start with "ab" , then followed by one more character.

Example: "abc", "abb", "abx" ....

"?bc" -All the lower level components which start with any one character ,then followed by "c".

Example: "abc", "xbc", "bbc" ....

"a?c" -All the lower level components which start with "a" , then followed by one more character and then followed by "c".

Example: "abc", "aac", "axc" ..

There are two ways to get the configuration data:

1)Automatic : Using Field macros

2)Manual : using gte\_config\_\* methods.

### **Automatic Configuration:**

To use the atomic configuration, all the configurable fields should be defined using ovm component field macros and ovm component utilities macros.

### **Ovm component utility macros:**

For non parameterized classes

```
`ovm_component_utils_begin(TYPE)
 `ovm_field_* macro invocations here
`ovm_component_utils_end
```

For parameterized classes.

```
`ovm_component_param_utils_begin(TYPE)
 `ovm_field_* macro invocations here
`ovm_component_utils_end
```

For OVM Field macros, Refer to link

### **OVM TRANSACTION**

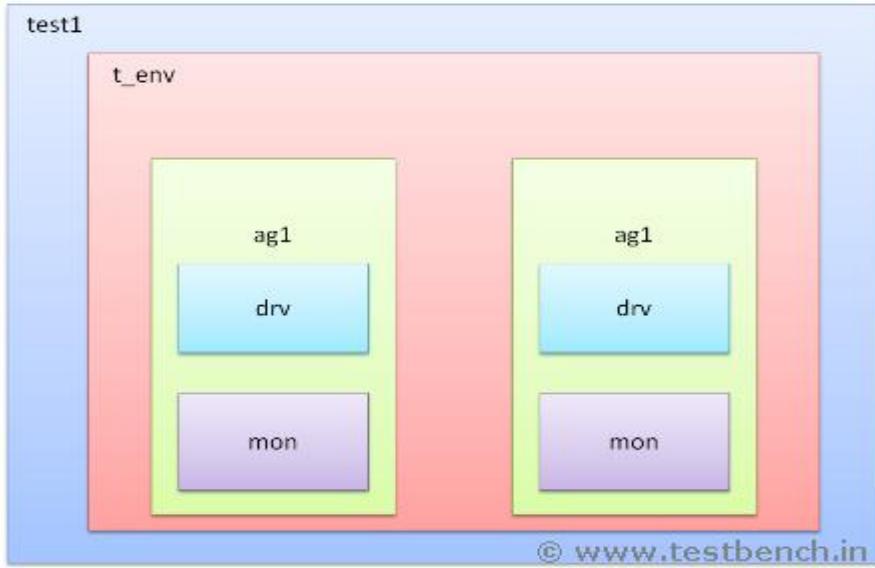
Example:

Following example is from link

### **OVM TESTBENCH**

2 Configurable fields, a integer and a string are defined in env, agent, monitor and driver classes.

Topology of the environment using these classes is



### Driver class Source Code:

Similar to driver class, all other components env, agent and monitor are define.

```

class driver extends ovm_driver;
  integer int_cfg;
  string str_cfg;
  `ovm_component_utils_begin(driver)
    `ovm_field_int(int_cfg, OVM_DEFAULT)
    `ovm_field_string(str_cfg, OVM_DEFAULT)
  `ovm_component_utils_end
  function new(string name, ovm_component parent);
    super.new(name, parent);
  endfunction
  function void build();
    super.build();
  endfunction
endclass

```

### Testcase:

Using set\_config\_int() and set\_config\_string() configure variables at various hierachal locations.

```

//t_env.ag1.drv.int_cfg
//t_env.ag1.mon.int_cfg
set_config_int("*.*","int_cfg",32);

//t_env.ag2.drv
set_config_int("t_env.ag2.drv","int_cfg",32);

```

```

//t_env.ag2.mon
set_config_int("t_env.ag2.mon","int_cfg",32);
/t_env.ag1.mon.str_cfg
/t_env.ag2.mon.str_cfg
/t_env.ag1.drv.str_cfg
/t_env.ag2.drv.str_cfg
set_config_string("*.*.ag?.*","str_cfg","pars");
/t_env.str_cfg
set_config_string("t_env","str_cfg","abcd");

```

### Download the source code

[ovm\\_configuration\\_1.tar](#)

[Browse the code in ovm\\_configuration\\_1.tar](#)

### Command to run the simulation

your\_tool\_simulation\_command +path\_to\_ovm\_pkg -f filelist +OVM\_TESTNAME=test1

From the above log report of th example, we can see the variables int\_cfg and str\_cfg of all the components and they are as per the configuration setting from the testcase.

### Manual Configurations:

Using get\_config\_\* methods, user can get the required data if the data is available in the table. Following are the method to get configure data of type integer , string and object of ovm\_object based class respectively.

```

function bit get_config_int (string field_name,
    inout ovm_bitstream_t value)
function bit get_config_string (string field_name,
    inout string value)
function bit get_config_object (string field_name,
    inout ovm_object value,
    input bit clone = 1)

```

If a entry is found in the table with "field\_name" then data will be updated to "value" argument . If entry is not found, then the function returns "0". So when these methods are called, check the return value.

Example:

### Driver class code:

```

class driver extends ovm_driver;
integer int_cfg;
string str_cfg;
`ovm_component_utils(driver)
function new(string name, ovm_component parent);
    super.new(name, parent);
endfunction

```

```

function void build();
    super.build();
    void'(get_config_int("int_cfg",int_cfg));
    void'(get_config_string("str_cfg",str_cfg));
    ovm_report_info(get_full_name(),
        $psprintf("int_cfg %0d : str_cfg %0s ",int_cfg,str_cfg),OVM_LOW);
endfunction
endclass

```

[Download the source code](#)

[ovm\\_configuration\\_2.tar](#)

[Browse the code in ovm\\_configuration\\_2.tar](#)

[Command to run the simulation](#)

your\_tool\_simulation\_command +path\_to\_ovm\_pkg -f filelist +OVM\_TESTNAME=test1

[Log file](#)

```

OVM_INFO @ 0: ovm_test_top.t_env
    int_cfg x : str_cfg abcd
OVM_INFO @ 0: ovm_test_top.t_env.ag1
    int_cfg x : str_cfg
OVM_INFO @ 0: ovm_test_top.t_env.ag1.drv
    int_cfg 32 : str_cfg pars
OVM_INFO @ 0: ovm_test_top.t_env.ag1.mon
    int_cfg 32 : str_cfg pars
OVM_INFO @ 0: ovm_test_top.t_env.ag2
    int_cfg x : str_cfg
OVM_INFO @ 0: ovm_test_top.t_env.ag2.drv
    int_cfg 32 : str_cfg pars
OVM_INFO @ 0: ovm_test_top.t_env.ag2.mon
    int_cfg 32 : str_cfg pars

```

[Configuration Setting Members:](#)

[print\\_config\\_settings](#)

```

function void print_config_settings
    (string field = """",
        ovm_component comp = null,
        bit recurse = 0)

```

This method prints all configuration information for this component.

If "field" is specified and non-empty, then only configuration settings matching that field, if any, are printed. The field may not contain wildcards. If "recurse" is set, then information for all children components are printed recursively.

[print\\_config\\_matches](#)

**static bit** print\_config\_matches = 0

Setting this static variable causes `get_config_*` to print info about matching configuration settings as they are being applied. These two members will be helpful to know while debugging.

**Download the source code**

[ovm\\_configuration\\_3.tar](#)

[Browse the code in ovm\\_configuration\\_3.tar](#)

**Command to run the simulation**

`your_tool_simulation_command +path_to_ovm_pkg -f filelist +OVM_TESTNAME=test1`

**Log file**

When `print_config_settings` method is called

`ovm_test_top.t_env.ag1.drv`

`ovm_test_top.*.ag1.* int_cfg int 32`

`ovm_test_top.t_env.ag1.drv.rsp_port`

`ovm_test_top.*.ag?.* str_cfg string pars`

`ovm_test_top.t_env.ag1.drv.rsp_port`

`ovm_test_top.*.ag1.* int_cfg int 32`

`ovm_test_top.t_env.ag1.drv.sqr_pull_port`

`ovm_test_top.*.ag?.* str_cfg string pars`

`ovm_test_top.t_env.ag1.drv.sqr_pull_port`

`ovm_test_top.*.ag1.* int_cfg int 32`

When `print_config_matches` is set to 1.

`OVM_INFO @ 0: ovm_test_top.t_env [auto-configuration]`

Auto-configuration matches for component `ovm_test_top.t_env` (env).

Last entry for a given field takes precedence.

Config set from	Instance Path	Field name	Type	Value
-----------------	---------------	------------	------	-------

---

ovm_test_top(test1)	<code>ovm_test_top.t_env</code>	<code>str_cfg</code>	string	abcd
---------------------	---------------------------------	----------------------	--------	------

## **INDEX**

### **INTRODUCTION**

#### **SPECIFICATION**

Switch Specification

Packet Format

Packet Header

Configuration

Interface Specification

Memory Interface

Input Port

Output Port

### **VERIFICATION PLAN**

Overview

Feature Extraction

Stimulus Generation Plan

Coverage Plan

Verification Environment

### **PHASE 1 TOP**

Interfaces

Testcase

Top Module

Top Module Source Code **PHASE 2 ENVIRONMENT**

Environment Class

Run

Environment Class Source Code

### **PHASE 3 RESET**

### **PHASE 4 PACKET**

Packet Class Source Code

Program Block Source Code **PHASE 5 DRIVER**

Driver Class Source Code

Environment Class Source Code

### **PHASE 6 RECEIVER**

Receiver Class Source Code

Environment Class Source Code **PHASE 7 SCOREBOARD**

Scoreboard Class Source Code

Source Code Of The Environment Class **PHASE 8 COVERAGE**

Source Code Of Coverage Class

Source Code Of The Scoreboard Class

### **PHASE 9 TESTCASE**

Source Code Of Constraint Testcase

## **INTRODUCTION**

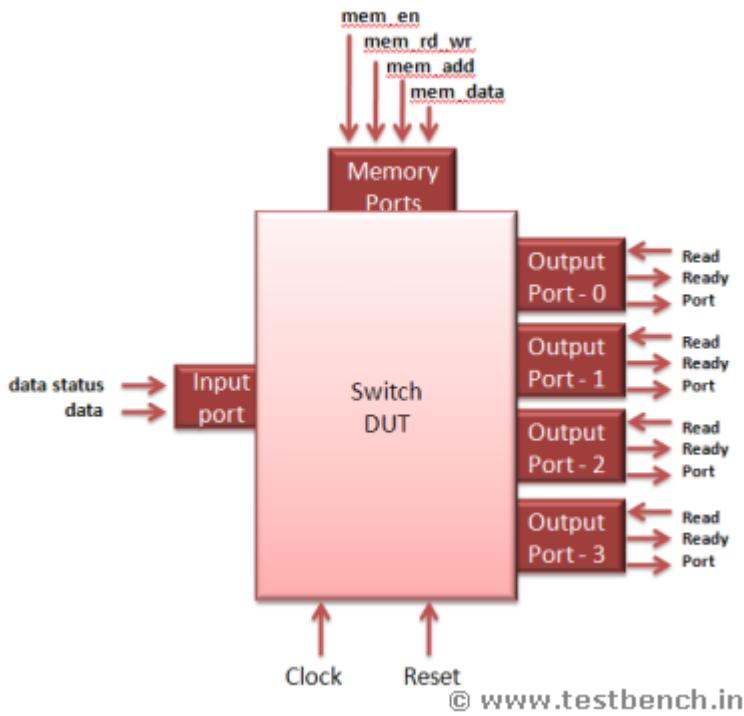
In this tutorial, we will verify the Switch RTL core. Following are the steps we follow to verify the Switch RTL core.

- 1) Understand the specification
- 2) Developing Verification Plan
- 3) Building the Verification Environment. We will build the Environment in Multiple phases, so it will be easy for you to lean step by step.
  - ⌚ Phase 1) We will develop the testcase and interfaces, and integrate them in these with the DUT in top module.
  - ⌚ Phase 2) We will Develop the Environment class.
  - ⌚ Phase 3) We will develop reset and configuration methods in Environment class. Then using these methods, we will reset the DUT and configure the port address.
  - ⌚ Phase 4) We will develop a packet class based on the stimulus plan. We will also write a small code to test the packet class implementation.
  - ⌚ Phase 5) We will develop a driver class. Packets are generated and sent to dut using driver.
  - ⌚ Phase 6) We will develop receiver class. Receiver collects the packets coming from the output port of the DUT.
  - ⌚ Phase 7) We will develop scoreboard class which does the comparison of the expected packet with the actual packet received from the DUT.
  - ⌚ Phase 8) We will develop coverage class based on the coverage plan.
  - ⌚ Phase 9) In this phase , we will write testcases and analyze the coverage report.

## **SPECIFICATION**

### **Switch Specification:**

This is a simple switch. Switch is a packet based protocol. Switch drives the incoming packet which comes from the input port to output ports based on the address contained in the packet. The switch has a one input port from which the packet enters. It has four output ports where the packet is driven out.



### Packet Format:

Packet contains 3 parts. They are Header, data and frame check sequence.

Packet width is 8 bits and the length of the packet can be between 4 bytes to 259 bytes.

### Packet Header:

Packet header contains three fields DA, SA and length.

② DA: Destination address of the packet is of 8 bits. The switch drives the packet to respective ports based on this destination address of the packets. Each output port has 8-bit unique port address. If the destination address of the packet matches the port address, then switch drives the packet to the output port.

② SA: Source address of the packet from where it originate. It is 8 bits.

Length: Length of the data is of 8 bits and from 0 to 255. Length is measured in terms of bytes.

If Length = 0, it means data length is 0 bytes

If Length = 1, it means data length is 1 bytes

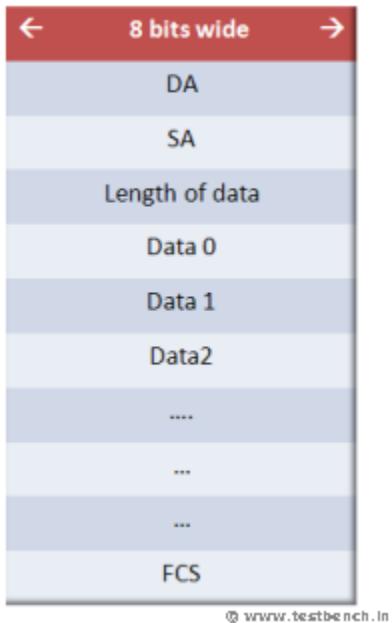
If Length = 2, it means data length is 2 bytes

If Length = 255, it means data length is 255 bytes

② Data: Data should be in terms of bytes and can take anything.

② FCS: Frame check sequence

This field contains the security check of the packet. It is calculated over the header and data.



### Configuration:

Switch has four output ports. These output ports address have to be configured to a unique address. Switch matches the DA field of the packet with this configured port address and sends the packet on to that port. Switch contains a memory. This memory has 4 locations, each can store 8 bits. To configure the switch port address, memory write operation has to be done using memory interface. Memory address (0,1,2,3) contains the address of port(0,1,2,3) respectively.

### Interface Specification:

The Switch has one input Interface, from where the packet enters and 4 output interfaces from where the packet comes out and one memory interface, through the port address can be configured. Switch also has a clock and asynchronous reset signal.

### Memory Interface:

Through memory interfaced output port address are configured. It accepts 8 bit data to be written to memory. It has 8 bit address inputs. Address 0,1,2,3 contains the address of the port 0,1,2,3 respectively.

There are 4 input signals to memory interface. They are

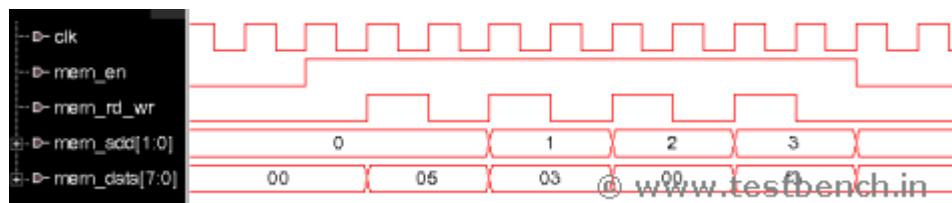
```
input mem_en;
input mem_rd_wr;
input [1:0] mem_add;
input [7:0] mem_data;
```

All the signals are active high and are synchronous to the positive edge of clock signal.

To configure a port address,

1. Assert the mem\_en signal.

2. Assert the mem\_rd\_wr signal.
3. Drive the port number (0 or 1 or 2 or 3) on the mem\_add signal
4. Drive the 8 bit port address on to mem\_data signal.



### Input Port

Packets are sent into the switch using input port.

All the signals are active high and are synchronous to the positive edge of clock signal.

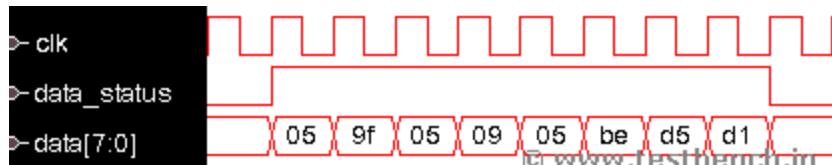
**input** port has **2 input** signals. They are

**input [7:0] data;**

**input** data\_status;

To send the packet in to switch,

1. Assert the data\_status signal.
2. Send the packet on the data signal byte by byte.
3. After sending all the data bytes, deassert the data\_status signal.
4. There should be at least 3 clock cycles difference between packets.



### Output Port

Switch sends the packets out using the output ports. There are 4 ports, each having data, ready and read signals. All the signals are active high and are synchronous to the positive edge of clock signal.

Signal list is

**output [7:0] port0;**

**output [7:0] port1;**

**output [7:0] port2;**

**output [7:0] port3;**

**output** ready\_0;

**output** ready\_1;

**output** ready\_2;

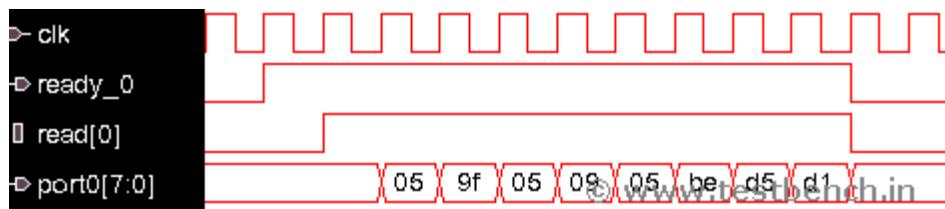
```

output ready_3;
input read_0;
input read_1;
input read_2;
input read_3;

```

When the data is ready to be sent out from the port, switch asserts ready\_\* signal high indicating that data is ready to be sent.

If the read\_\* signal is asserted, when ready\_\* is high, then the data comes out of the port\_\* signal after one clock cycle.



### RTL code:

RTL code is attached with the tar files. From the Phase 1, you can download the tar files.

## **VERIFICATION PLAN**

### Overview

This Document describes the Verification Plan for Switch. The Verification Plan is based on System Verilog Hardware Verification Language. The methodology used for Verification is Constraint random coverage driven verification.

### Feature Extraction

This section contains list of all the features to be verified.

1)

ID: Configuration

Description: Configure all the 4 port address with unique values.

2)

ID: Packet DA

Description: DA field of packet should be any of the port address. All the 4 port address should be used.

3)

ID : Packet payload

Description: Length can be from 0 to 255. Send packets with all the lengths.

4)

ID: Length

Description:

Length field contains length of the payload.

Send Packet with correct length field and incorrect length fields.

5)

ID: FCS

Description:

Good FCS: Send packet with good FCS.

Bad FCS: Send packet with corrupted FCS.

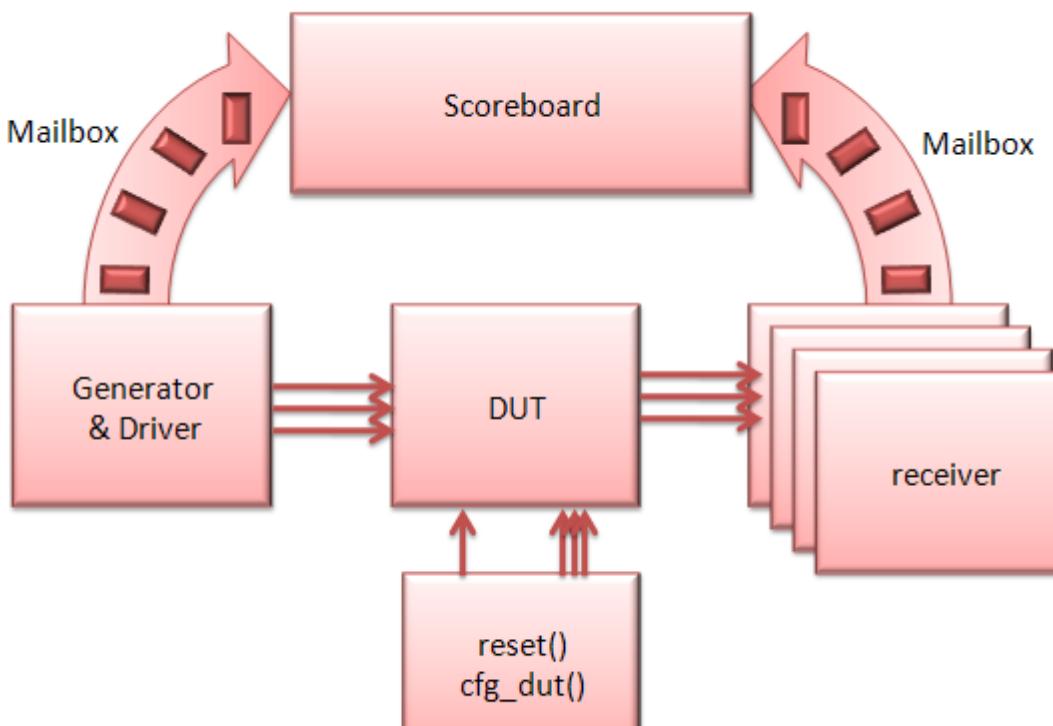
### **Stimulus Generation Plan**

- 1) Packet DA: Generate packet DA with the configured address.
- 2) Payload length: generate payload length ranging from 2 to 255.
- 3) Correct or Incorrect Length field.
- 4) Generate good and bad FCS.

### **Coverage Plan**

- 1) Cover all the port address configurations.
- 2) Cover all the packet lengths.
- 3) Cover all correct and incorrect length fields.
- 4) Cover good and bad FCS.
- 5) Cover all the above combinations.

### **Verification Environment**



## PHASE 1 TOP

In phase 1,

- 1) We will write SystemVerilog Interfaces for input port, output port and memory port.
- 2) We will write Top module where testcase and DUT instances are done.
- 3) DUT and TestBench interfaces are connected in top module.
- 4) Clock is generator in top module.

**NOTE:** In every file you will see the syntax

```
`ifndef GUARD_*
`endif GUARD_*
```

## Interfaces

In the interface.sv file, declare the 3 interfaces in the following way.

- ⌚ All the interfaces has clock as input.
- ⌚ All the signals in interface are logic type.
- ⌚ All the signals are synchronized to clock except reset in clocking block.
- ⌚ Signal directional w.r.t TestBench is specified with modport.

```
`ifndef GUARD_INTERFACE
`define GUARD_INTERFACE
///////////////////////////////
// Interface declaration for the memory///
///////////////////////////////

interface mem_interface(input bit clock);
    logic [7:0] mem_data;
    logic [1:0] mem_add;
    logic      mem_en;
    logic      mem_rd_wr;

    clocking cb@(posedge clock);
        default input #1 output #1;
        output   mem_data;
        output   mem_add;
        output   mem_en;
        output   mem_rd_wr;
    endclocking
    modport MEM(clocking cb,input clock);
endinterface

/////////////////////////////
// Interface for the input side of switch.//
// Reset signal is also passed hear.    //
/////////////////////////////
```

```

interface input_interface(input bit clock);
    logic      data_status;
    logic [7:0] data_in;
    logic      reset;

    clocking cb@(posedge clock);
        default input #1 output #1;
        output   data_status;
        output   data_in;
    endclocking

    modport IP(clocking cb,output reset,input clock);

endinterface

///////////////////////////////
// Interface for the output side of the switch.//
// output_interface is for only one output port//
///////////////////////////////

interface output_interface(input bit clock);
    logic [7:0] data_out;
    logic ready;
    logic read;

    clocking cb@(posedge clock);
        default input #1 output #1;
        input    data_out;
        input    ready;
        output   read;
    endclocking
    modport OP(clocking cb,input clock);
endinterface

/////////////////////////////
`endif

```

## Testcase

Testcase is a program block which provides an entry point for the test and creates a scope that encapsulates program-wide data. Currently this is an empty testcase which just ends the simulation after 100 time units. Program block contains all the above declared interfaces as arguments. This testcase has initial and final blocks.

```
`ifndef GUARD_TESTCASE
`define GUARD_TESTCASE
program testcase(mem_interface.MEM mem_intf,input_interface.IP input_intf,output_interface.OP output_intf[4]);
initial
begin
$display(" ***** Start of testcase *****");
#1000;
end
final
$display(" ***** End of testcase *****");
endprogram
`endif
```

## Top Module

The modules that are included in the source text but are not instantiated are called top modules. This module is the highest scope of modules. Generally this module is named as "top" and referenced as "top module". Module name can be anything.

Do the following in the top module:

1)Generate the clock signal.

```
bit Clock;
```

initial

```
  forever #10 Clock = ~Clock;
```

2)Do the instances of memory interface.

```
mem_interface mem_intf(Clock);
```

3)Do the instances of input interface.

```
input_interface input_intf(Clock);
```

4)There are 4 output ports. So do 4 instances of output\_interface.

```
output_interface output_intf[4](Clock);
```

5)Do the instance of testcase and pass all the above declared interfaces.

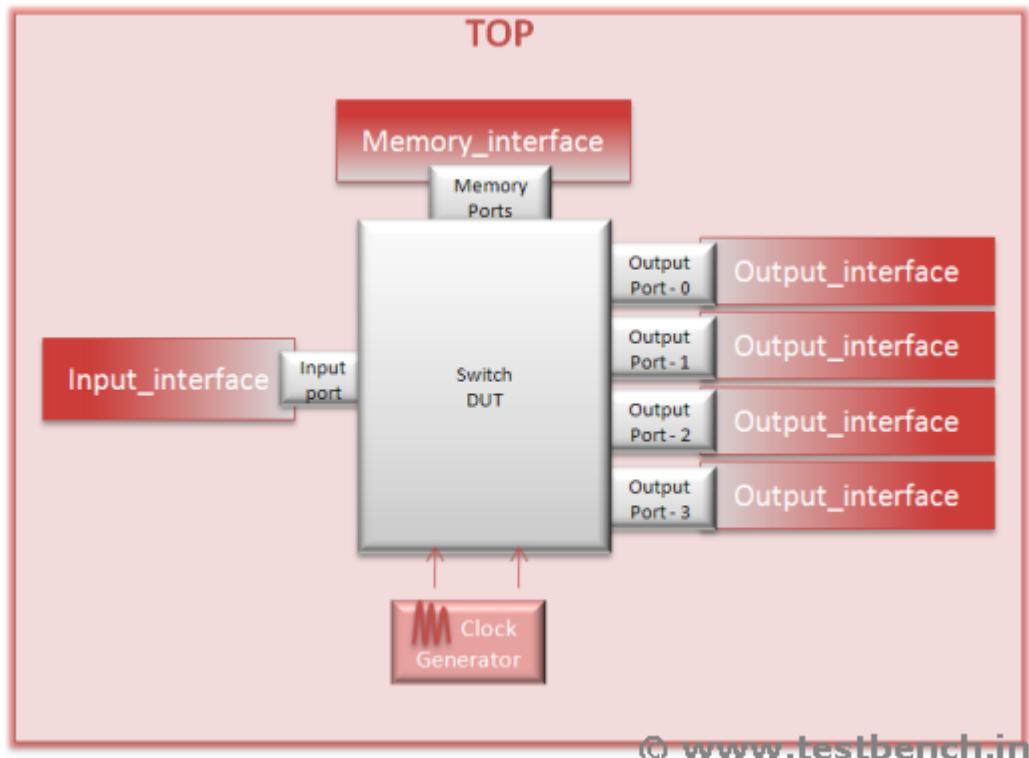
```
testcase TC (mem_intf,input_intf,output_intf);
```

6)Do the instance of DUT.

```
switch DUT (.
```

7)Connect all the interfaces and DUT. The design which we have taken is in verilog. So Verilog DUT instance is connected signal by signal.

```
switch DUT (.clk(Clock),
    .reset(input_intf.reset),
    .data_status(input_intf.data_status),
    .data(input_intf.data_in),
    .port0(output_intf[0].data_out),
    .port1(output_intf[1].data_out),
    .port2(output_intf[2].data_out),
    .port3(output_intf[3].data_out),
    .ready_0(output_intf[0].ready),
    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),
    .mem_en(mem_intf.mem_en),
    .mem_rd_wr(mem_intf.mem_rd_wr),
    .mem_add(mem_intf.mem_add),
    .mem_data(mem_intf.mem_data));
```



## Top Module Source Code:

```
ifndef GUARD_TOP
#define GUARD_TOP

module top();
// Clock Declaration and Generation
// bit Clock;
initial
forever #10 Clock = ~Clock;
// Memory interface instance
// mem#(int) mem_ifc(Clock);
mem_ifc mem_intf(Clock);
```

```

/////////
// Input interface instance          //
/////////

input_interface input_intf(Clock);

/////////
// output interface instance         //
/////////

output_interface output_intf[4](Clock);

/////////
// Program block Testcase instance   //
/////////

testcase TC (mem_intf,input_intf,output_intf);

/////////
// DUT instance and signal connection //
/////////

switch DUT (.clk(Clock),
    .reset(input_intf.reset),
    .data_status(input_intf.data_status),
    .data(input_intf.data_in),
    .port0(output_intf[0].data_out),
    .port1(output_intf[1].data_out),
    .port2(output_intf[2].data_out),
    .port3(output_intf[3].data_out),
    .ready_0(output_intf[0].ready),
    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),
    .mem_en(mem_intf.mem_en),

```

```
.mem_rd_wr(mem_intf.mem_rd_wr),  
.mem_add(mem_intf.mem_add),  
.mem_data(mem_intf.mem_data));
```

**endmodule**

`endif

**Download the phase 1 files:**

switch\_1.tar

Browse the code in switch\_1.tar

**Run the simulation:**

```
vcs -sverilog -f filelist -R -ntb_opts dtm
```

**Log file after simulation:**

```
***** Start of testcase *****  
***** End of testcase *****
```

## **PHASE 2 ENVIRONMENT**

In this phase, we will write

- ⌚ Environment class.
- ⌚ Virtual interface declaration.
- ⌚ Defining Environment class constructor.
- ⌚ Defining required methods for execution. Currently these methods will not be implemented in this phase.

All the above are done in Environment.sv file.

We will write a testcase using the above define environment class in testcase.sv file.

### **Environment Class:**

The class is a base class used to implement verification environments. Testcase contains the instance of the environment class and has access to all the public declaration of environment class.

All methods are declared as virtual methods. In environment class, we will formalize the simulation steps using virtual methods. The methods are used to control the execution of the simulation.

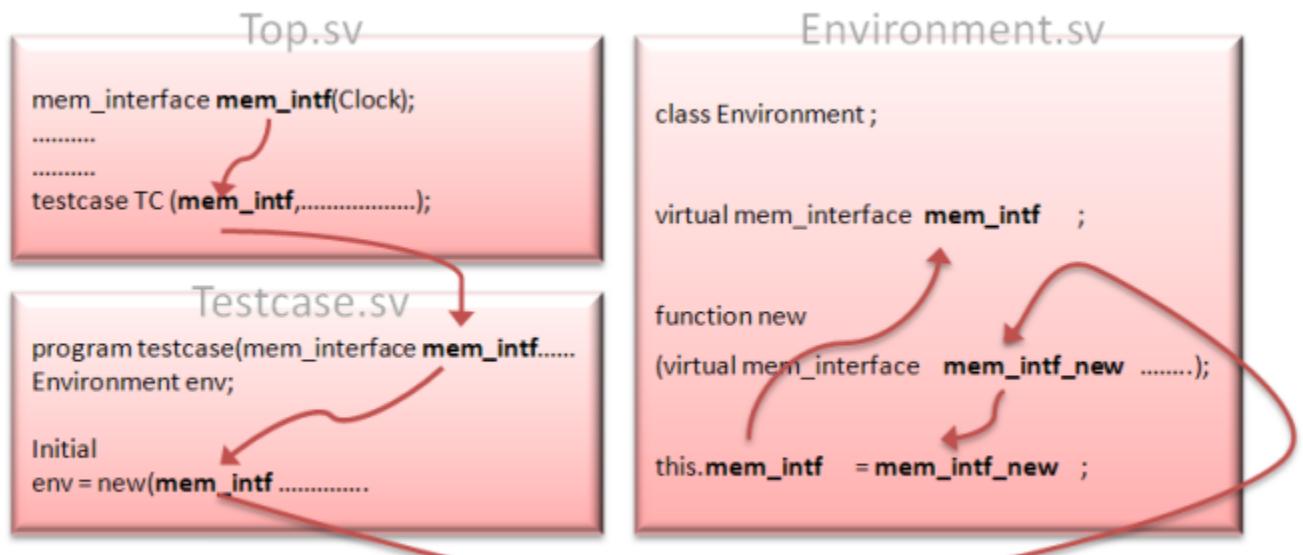
Following are the methods which are going to be defined in environment class.

- 1) new() : In constructor method, we will connect the virtual interfaces which are passed as argument to the virtual interfaces to those which are declared in environment class.
- 2) build(): In this method , all the objects like driver, monitor etc are constructed. Currently this method is empty as we did not develop any other component.
- 3) reset(): in this method we will reset the DUT.
- 4) cfg\_dut(): In this method, we will configure the DUT output port address.

- 5) start(): in this method, we will call the methods which are declared in the other components like driver and monitor.
- 6) wait\_for\_end(): this method is used to wait for the end of the simulation. Waits until all the required operations in other components are done.
- 7) report(): This method is used to print the TestPass and TestFail status of the simulation, based on the error count..
- 8) run(): This method calls all the above declared methods in a sequence order. The testcase calls this method, to start the simulation.

We are not implementing build(), reset(), cfg\_dut() , strat() and report() methods in this phase. Connecting the virtual interfaces of Environment class to the physical interfaces of top module. Verification environment contains the declarations of the virtual interfaces. Virtual interfaces are just a handles(like pointers). When a virtual interface is declared, it only creates a handle. It doesnot creat a real interface.

Constructor method should be declared with virtual interface as arguments, so that when the object is created in testcase, new() method can pass the interfaces in to environment class where they are assigned to the local virtual interface handle. With this, the Environment class virtual interfaces are pointed to the physical interfaces which are declared in the top module.



© www.testbench.in

Declare virtual interfaces in Environment class.

```

virtual mem_interface.MEM mem_intf ;
virtual input_interface.IP input_intf ;
virtual output_interface.OP output_intf[4] ;
```

The construction of Environment class is declared with virtual interface as arguments.

```
function new(virtual mem_interface.MEM mem_intf_new ,  
           virtual input_interface.IP input_intf_new ,  
           virtual output_interface.OP output_intf_new[4] );
```

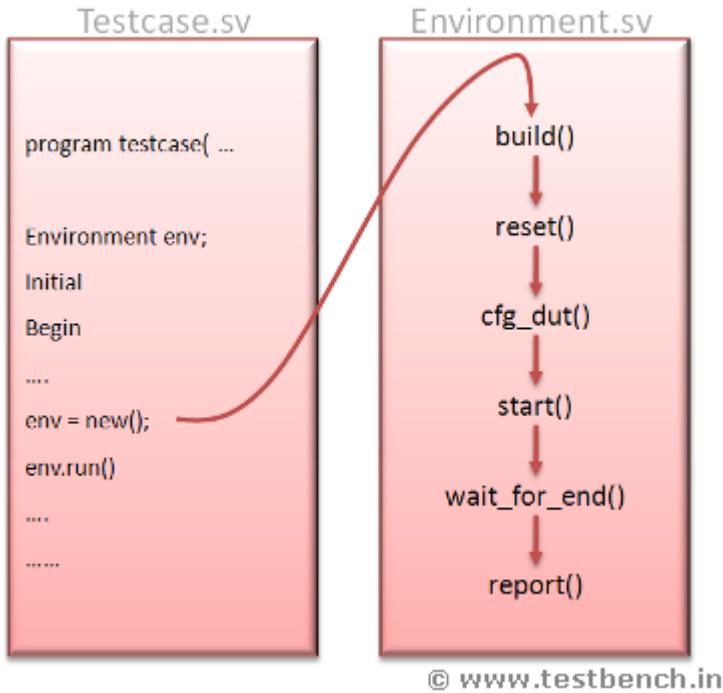
In constructor methods, the interfaces which are arguments are connected to the virtual interfaces of environment class.

```
this.mem_intf = mem_intf_new ;  
this.input_intf = input_intf_new ;  
this.output_intf = output_intf_new ;
```

#### Run :

The run() method is called from the testcase to start the simulation. run() method calls all the methods which are defined in the Environment class.

```
task run();  
$display("%0d : Environment : start of run() method",$time);  
build();  
reset();  
cfg_dut();  
start();  
wait_for_end();  
report();  
$display("%0d : Environment : end of run() method",$time);  
endtask : run
```



### Environment Class Source Code:

```

`ifndef GUARD_ENV
`define GUARD_ENV
class Environment;

    virtual mem_interface.MEM mem_intf    ;
    virtual input_interface.IP input_intf   ;
    virtual output_interface.OP output_intf[4] ;

function new(virtual mem_interface.MEM mem_intf_new    ,
            virtual input_interface.IP input_intf_new   ,
            virtual output_interface.OP output_intf_new[4]);
    this.mem_intf    = mem_intf_new    ;
    this.input_intf   = input_intf_new   ;
    this.output_intf  = output_intf_new ;
    $display("%0d : Environment : created env object",$time);
endfunction : new
function void build();
    $display("%0d : Environment : start of build() method",$time);
    $display("%0d : Environment : end of build() method",$time);
endfunction : build

```

```

task reset();
$display("%0d : Environment : start of reset() method",$time);
$display("%0d : Environment : end of reset() method",$time);
endtask : reset

task cfg_dut();
$display("%0d : Environment : start of cfg_dut() method",$time);
$display("%0d : Environment : end of cfg_dut() method",$time);
endtask : cfg_dut

task start();
$display("%0d : Environment : start of start() method",$time);
$display("%0d : Environment : end of start() method",$time);
endtask : start

task wait_for_end();
$display("%0d : Environment : start of wait_for_end() method",$time);
$display("%0d : Environment : end of wait_for_end() method",$time);
endtask : wait_for_end

task run();
$display("%0d : Environment : start of run() method",$time);
build();
reset();
cfg_dut();
start();
wait_for_end();
report();
$display("%0d : Environment : end of run() method",$time);
endtask : run

task report();
endtask : report

endclass

`endif

```

We will create a file Global.sv for global requirement. In this file, define all the port address as macros in this file. Define a variable error as integer to keep track the number of errors occurred during the simulation.

```

`ifndef GUARD_GLOBALS
`define GUARD_GLOBALS
`define P0 8'h00
`define P1 8'h11
`define P2 8'h22
`define P3 8'h33
int error = 0;
int num_of_pkts = 10;
`endif

```

Now we will update the testcase. Take an instance of the Environment class and call the run method of the Environment class.

```

`ifndef GUARD_TESTCASE
`define GUARD_TESTCASE
program testcase(mem_interface.MEM mem_intf,input_interface.IP
input_intf,output_interface.OP output_intf[4]);
Environment env;
initial
begin
$display(" ***** Start of testcase *****");
env = new(mem_intf,input_intf,output_intf);
env.run();
#1000;
end
final
$display(" ***** End of testcase *****");
endprogram
`endif

```

#### Download the phase 2 source code:

[switch\\_2.tar](#)

[Browse the code in switch\\_2.tar](#)

#### Run the simulation:

vcs -sverilog -f filelist -R -ntb\_opts dtm

#### Log report after the simulation:

```

***** Start of testcase *****
0 : Environemnt : created env object
0 : Environemnt : start of run() method
0 : Environemnt : start of build() method
0 : Environemnt : end of build() method
0 : Environemnt : start of reset() method

```

```

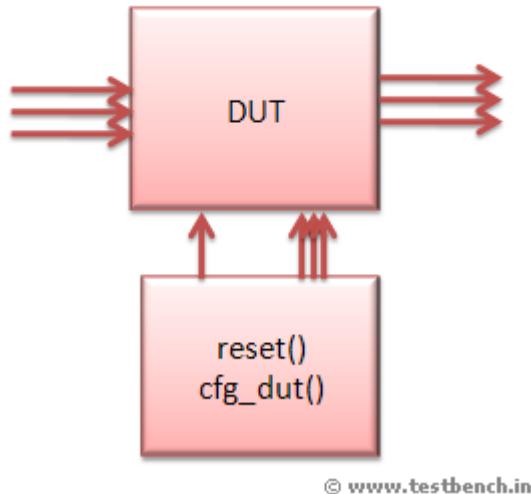
0 : Environemnt : end of reset() method
0 : Environemnt : start of cfg_dut() method
0 : Environemnt : end of cfg_dut() method
0 : Environemnt : start of start() method
0 : Environemnt : end of start() method
0 : Environemnt : start of wait_for_end() method
0 : Environemnt : end of wait_for_end() method
0 : Environemnt : end of run() method
***** End of testcase *****

```

### **PHASE 3 RESET**

In this phase we will reset and configure the DUT.

The Environment class has reset() method which contains the logic to reset the DUT and cfg\_dut() method which contains the logic to configure the DUT port address.



**NOTE:** Clocking block signals can be driven only using a non-blocking assignment.

Define the reset() method.

1) Set all the DUT input signals to a known state.

```

mem_intf.cb.mem_data    <= 0;
mem_intf.cb.mem_add     <= 0;
mem_intf.cb.mem_en      <= 0;
mem_intf.cb.mem_rd_wr   <= 0;
input_intf.cb.data_in   <= 0;
input_intf.cb.data_status <= 0;
output_intf[0].cb.read   <= 0;

```

```

output_intf[1].cb.read  <= 0;
output_intf[2].cb.read  <= 0;
output_intf[3].cb.read  <= 0;

2) Reset the DUT.
// Reset the DUT
input_intf.reset  <= 1;
repeat (4) @ input_intf.clock;
input_intf.reset  <= 0;

3) Updated the cfg_dut method.
task cfg_dut();
$display("%0d : Environment : start of cfg_dut() method",$time);
mem_intf.cb.mem_en <= 1;
@(posedge mem_intf.clock);
mem_intf.cb.mem_rd_wr <= 1;
@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h0;
mem_intf.cb.mem_data <= `P0;
$display("%0d : Environment : Port 0 Address %h",$time,`P0);
@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h1;
mem_intf.cb.mem_data <= `P1;
$display("%0d : Environment : Port 1 Address %h",$time,`P1);
@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h2;
mem_intf.cb.mem_data <= `P2;
$display("%0d : Environment : Port 2 Address %h",$time,`P2);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h3;
mem_intf.cb.mem_data <= `P3;
$display("%0d : Environment : Port 3 Address %h",$time,`P3);

@(posedge mem_intf.clock);
mem_intf.cb.mem_en <= 0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add <= 0;
mem_intf.cb.mem_data <= 0;

$display("%0d : Environment : end of cfg_dut() method",$time);
endtask : cfg_dut

```

(4) In wait\_for\_end method, wait for some clock cycles.

```
task wait_for_end();
$display("%0d : Environment : start of wait_for_end() method",$time);
repeat(10000) @(input_intf.clock);
$display("%0d : Environment : end of wait_for_end() method",$time);
endtask : wait_for_end
```

[Download the Phase 3 source code:](#)

[switch\\_3.tar](#)

[Browse the code in switch\\_3.tar](#)

[Run the simulation:](#)

```
vcs -sverilog -f filelist -R -ntb_opts dtm
```

[Log File report](#)

```
***** Start of testcase *****
0 : Environment : created env object
0 : Environment : start of run() method
0 : Environment : start of build() method
0 : Environment : end of build() method
0 : Environment : start of reset() method
40 : Environment : end of reset() method
40 : Environment : start of cfg_dut() method
70 : Environment : Port 0 Address 00
90 : Environment : Port 1 Address 11
110 : Environment : Port 2 Address 22
130 : Environment : Port 3 Address 33
150 : Environment : end of cfg_dut() method
150 : Environment : start of start() method
150 : Environment : end of start() method
150 : Environment : start of wait_for_end() method
100150 : Environment : end of wait_for_end() method
100150 : Environment : end of run() method
***** End of testcase *****
```

## **PHASE 4 PACKET**

In this Phase, We will define a packet and then test it whether it is generating as expected. Packet is modeled using class. Packet class should be able to generate all possible packet types randomly. Packet class should also implement required methods like packing(), unpacking(), compare() and display() methods.

We will write the packet class in packet.sv file. Packet class variables and constraints have been derived from stimulus generation plan.

Revisit Stimulus Generation Plan

- 1) Packet DA: Generate packet DA with the configured address.
- 2) Payload length: generate payload length ranging from 2 to 255.
- 3) Correct or Incorrect Length field.
- 4) Generate good and bad FCS.
  - 1) Declare FCS types as enumerated data types. Name members as GOOD\_FCS and BAD\_FCS.  
**typedef enum { GOOD\_FCS, BAD\_FCS } fcs\_kind\_t;**
  - 2) Declare the length type as enumerated data type. Name members as GOOD\_LENGTH and BAD\_LENGTH.  
**typedef enum { GOOD\_LENGTH, BAD\_LENGTH } length\_kind\_t;**
  - 3) Declare the length type and fcs type variables as rand.

```
rand fcs_kind_t   fcs_kind;  
rand length_kind_t length_kind;
```

- 4) Declare the packet field as rand. All fields are bit data types. All fields are 8 bit packet array. Declare the payload as dynamic array.

```
rand bit [7:0] length;  
rand bit [7:0] da;  
rand bit [7:0] sa;  
rand byte data[];//Payload using Dynamic array,size is generated on the fly  
rand byte fcs;
```

- 5) Constraint the DA field to be any one of the configured address.

```
constraint address_c { da inside {`P0,`P1,`P2,`P3} ; }
```

- 6) Constrain the payload dynamic array size to between 1 to 255.

```
constraint payload_size_c { data.size inside { [1 : 255]}; }
```

- 7) Constrain the payload length to the length field based on the length type.

```
constraint length_kind_c {  
    (length_kind == GOOD_LENGTH) -> length == data.size;  
    (length_kind == BAD_LENGTH) -> length == data.size + 2 ; }
```

Use solve before to direct the randomization to generate first the payload dynamic array size and then randomize length field.

```
constraint solve_size_length { solve data.size before length; }
```

8) Constrain the FCS field initial value based on the fcs kind field.

```
constraint fcs_kind_c {
    (fcs_kind == GOOD_FCS) -> fcs == 8'b0;
    (fcs_kind == BAD_FCS) -> fcs == 8'b1; }
```

9) Define the FCS method.

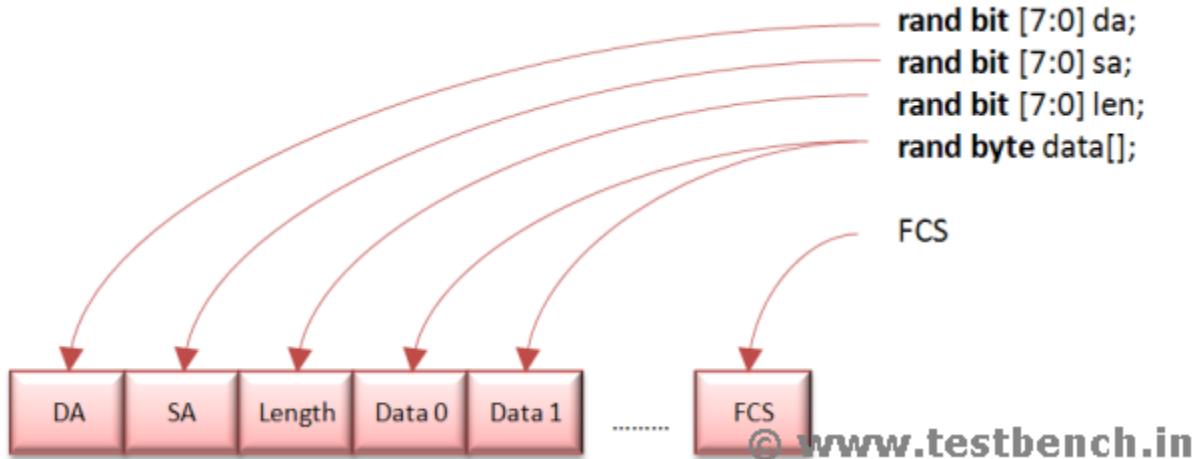
```
function byte cal_fcs;
    integer i;
    byte result ;
    result = 0;
    result = result ^ da;
    result = result ^ sa;
    result = result ^ length;
    for (i=0;i<data.size;i++)
        result = result ^ data[i];
    result = fcs ^ result;
    return result;
endfunction : cal_fcs
```

10) Define display methods:

Display method displays the current value of the packet fields to standard output.

```
virtual function void display();
    $display("\n----- PACKET KIND ----- ");
    $display(" fcs_kind : %s ",fcs_kind.name());
    $display(" length_kind : %s ",length_kind.name());
    $display("----- PACKET ----- ");
    $display(" 0 : %h ",da);
    $display(" 1 : %h ",sa);
    $display(" 2 : %h ",length);
    foreach(data[i])
        $write("%3d : %0h ",i+3,data[i]);
    $display("\n %2d : %h ",data.size() + 3 , cal_fcs);
    $display("----- \n");
endfunction : display
```

11) Define pack method:



Packing is commonly used to convert the high level data to low level data that can be applied to DUT. In packet class various fields are generated. Required fields are concatenated to form a stream of bytes which can be driven conveniently to DUT interface by the driver.

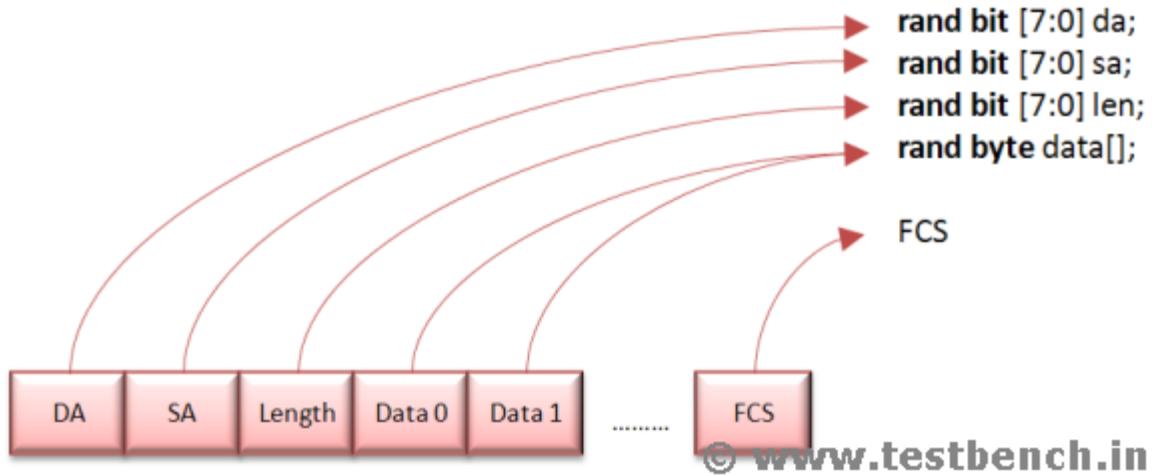
```

virtual function int unsigned byte_pack(ref logic [7:0] bytes[]);

bytes = new[data.size + 4];
bytes[0] = da;
bytes[1] = sa;
bytes[2] = length;
foreach(data[i])
  bytes[3 + i] = data[i];
bytes[data.size() + 3] = cal_fcs;
byte_pack = bytes.size;
endfunction : byte_pack

```

12) Define unpack method:



The unpack() method does exactly the opposite of pack method. Unpacking is commonly used to convert a data stream coming from DUT to high level data packet object.

```
virtual function void byte_unpack(const ref logic [7:0] bytes[]);  
    this.da = bytes[0];  
    this.sa = bytes[1];  
    this.length = bytes[2];  
    this.fcs = bytes[bytes.size - 1];  
    this.data = new[bytes.size - 4];  
    foreach(data[i])  
        data[i] = bytes[i + 3];  
    this.fcs = 0;  
    if(bytes[bytes.size - 1] != cal_fcs)  
        this.fcs = 1;  
endfunction : byte_unpack
```

14) Define a compare method.

Compares the current value of the object instance with the current value of the specified object instance.

If the value is different, FALSE is returned.

```
virtual function bit compare(packet pkt);  
    compare = 1;  
    if(pkt == null)  
        begin  
            $display(" ** ERROR ** : pkt : received a null object ");  
            compare = 0;  
        end  
    else  
        begin  
            if(pkt.da !== this.da)  
                begin  
                    $display(" ** ERROR **: pkt : Da field did not match");  
                    compare = 0;  
                end  
            if(pkt.sa !== this.sa)  
                begin  
                    $display(" ** ERROR **: pkt : Sa field did not match");  
                    compare = 0;  
                end  
        end  
endfunction : compare
```

```

if(pkt.length != this.length)
begin
    $display(" ** ERROR **: pkt : Length field did not match");
    compare = 0;
end
foreach(this.data[i])
if(pkt.data[i] != this.data[i])
begin
    $display(" ** ERROR **: pkt : Data[%0d] field did not match",i);
    compare = 0;
end

if(pkt.fcs != this.fcs)
begin
    $display(" ** ERROR **: pkt : fcs field did not match %h %h",pkt.fcs ,this.fcs);
    compare = 0;
end
end
endfunction : compare

```

### Packet Class Source Code

```

`ifndef GUARD_PACKET
`define GUARD_PACKET

//Define the enumerated types for packet types
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;
typedef enum { GOOD_LENGTH, BAD_LENGTH } length_kind_t;

class packet;
rand fcs_kind_t   fcs_kind;
rand length_kind_t length_kind;

rand bit [7:0] length;
rand bit [7:0] da;
rand bit [7:0] sa;
rand byte data[];//Payload using Dynamic array,size is generated on the fly
rand byte fcs;

```

```

constraint address_c { da inside {`P0,`P1,`P2,`P3} ; }

constraint payload_size_c { data.size inside { [1 : 255]};}

constraint length_kind_c {
    (length_kind == GOOD_LENGTH) -> length == data.size;
    (length_kind == BAD_LENGTH) -> length == data.size + 2 ;
}

constraint solve_size_length { solve data.size before length; }

constraint fcs_kind_c {
    (fcs_kind == GOOD_FCS) -> fcs == 8'b0;
    (fcs_kind == BAD_FCS) -> fcs == 8'b1; }

//// method to calculate the fcs /////
function byte cal_fcs;
    integer i;
    byte result ;
    result = 0;
    result = result ^ da;
    result = result ^ sa;
    result = result ^ length;
    for (i=0;i< data.size;i++)
        result = result ^ data[i];
    result = fcs ^ result;
    return result;
endfunction : cal_fcs

//// method to print the packet fields ///
virtual function void display();
    $display("\n----- PACKET KIND ----- ");
    $display(" fcs_kind : %s ",fcs_kind.name() );
    $display(" length_kind : %s ",length_kind.name() );
    $display("----- PACKET ----- ");
    $display(" 0 : %h ",da);
    $display(" 1 : %h ",sa);
    $display(" 2 : %h ",length);
    foreach(data[i])
        $write("%3d : %0h ",i+3,data[i]);
    $display("\n %2d : %h ",data.size() + 3 , cal_fcs);

```

```

$display("-----\n");
endfunction : display

//// method to pack the packet into bytes////
virtual function int unsigned byte_pack(ref logic [7:0] bytes[]);

bytes = new[data.size + 4];
bytes[0] = da;
bytes[1] = sa;
bytes[2] = length;
foreach(data[i])
bytes[3 + i] = data[i];
bytes[data.size() + 3] = cal_fcs;
byte_pack = bytes.size;
endfunction : byte_pack

////method to unpack the bytes in to packet /////
virtual function void byte_unpack(const ref logic [7:0] bytes[]);

this.da = bytes[0];
this.sa = bytes[1];
this.length = bytes[2];
this.fcs = bytes[bytes.size - 1];
this.data = new[bytes.size - 4];
foreach(data[i])
data[i] = bytes[i + 3];
this.fcs = 0;
if(bytes[bytes.size - 1] != cal_fcs)
this.fcs = 1;
endfunction : byte_unpack

//// method to compare the packets /////
virtual function bit compare(packet pkt);

compare = 1;
if(pkt == null)
begin
$display(" ** ERROR ** : pkt : received a null object ");
compare = 0;
end
else
begin
if(pkt.da != this.da)

```

```

begin
    $display(" ** ERROR **: pkt : Da field did not match");
    compare = 0;
end
if(pkt.sa != this.sa)
begin
    $display(" ** ERROR **: pkt : Sa field did not match");
    compare = 0;
end

if(pkt.length != this.length)
begin
    $display(" ** ERROR **: pkt : Length field did not match");
    compare = 0;
end
foreach(this.data[i])
if(pkt.data[i] != this.data[i])
begin
    $display(" ** ERROR **: pkt : Data[%0d] field did not match",i);
    compare = 0;
end

if(pkt.fcs != this.fcs)
begin
    $display(" ** ERROR **: pkt : fcs field did not match %h %h",pkt.fcs ,this.fcs);
    compare = 0;
end
end
endfunction : compare

endclass

```

Now we will write a small program to test our packet implantation. This program block is not used to verify the DUT.

Write a simple program block and do the instance of packet class. Randomize the packet and call the display method to analyze the generation. Then pack the packet in to bytes and then unpack bytes and then call compare method to check all the methods.

## Program Block Source Code

```
program test;

packet pkt1 = new();
packet pkt2 = new();
logic [7:0] bytes[];
initial
repeat(10)
if(pkt1.randomize)
begin
$display(" Randomization Successes full.");
pkt1.display();
void'(pkt1.byte_pack(bytes));
pkt2 = new();
pkt2.byte_unpack(bytes);
if(pkt2.compare(pkt1))
$display(" Packing,Unpacking and compare worked");
else
$display(" *** Something went wrong in Packing or Unpacking or compare ***");
end
else
$display(" *** Randomization Failed ***");
endprogram
```

## Download the packet class with program block.

[switch\\_4.tar](#)

[Browse the code in switch\\_4.tar](#)

## Run the simulation

vcs -sverilog -f filelist -R -ntb\_opts dtm

## Log file report:

Randomization Sucessesfull.

```

----- PACKET KIND -----
fcs_kind : BAD_FCS
length_kind : GOOD_LENGTH
----- PACKET -----
0 : 00
1 : f7
2 : be
3 : a6 4 : 1b 5 : b5 6 : fa 7 : 4e 8 : 15 9 : 7d 10 : 72 11 : 96 12 : 31 13 : c4 14 : aa 15 :
c4 16 : cf 17 : 4f 18 : f4 19 : 17 20 : 88 21 : f1 22 : 2c 23 : ce 24 : 5 25 : cb 26 : 8c 27 :
1a 28 : 37 29 : 60 30 : 5f 31 : 7a 32 : a2 33 : f0 34 : c9 35 : dc 36 : 41 37 : 3f 38 : 12 39 :
f4 40 : df 41 : c5 42 : d7 43 : 94 44 : 88 45 : 1 46 : 31 47 : 29 48 : d6 49 : f4 50 : d9 51 :
4f 52 : 0 53 : dd 54 : d2 55 : a6 56 : 59 57 : 43 58 : 45 59 : f2 60 : a2 61 : a1 62 : fd 63 :
ea 64 : c1 65 : 20 66 : c7 67 : 20 68 : e1 69 : 97 70 : c6 71 : cf 72 : cd 73 : 17 74 : 99 75 :
49 76 : b8 77 : 1c 78 : df 79 : e6 80 : 1a 81 : ce 82 : 8c 83 : ec 84 : b6 85 : bb 86 : a5 87 :
17 88 : cb 89 : 32 90 : e1 91 : 83 92 : 96 93 : e 94 : ee 95 : 57 96 : 33 97 : cd 98 : 62 99 :
88 100 : 7b 101 : e6 102 : 41 103 : ad 104 : 26 105 : ee 106 : 9c 107 : 95 108 : a7 109 : b8 110 :
83 111 : f 112 : ca 113 : ec 114 : b5 115 : 8d 116 : d8 117 : 2f 118 : 6f 119 : ea 120 : 4c 121 : 35
122 : 41 123 : f2 124 : 4e 125 : 89 126 : d8 127 : 78 128 : f1 129 : d 130 : d6 131 : d5 132 : 8 133 :
:c 134 : de 135 : a9 136 : 1d 137 : a0 138 : ae 139 : 99 140 : f5 141 : 53 142 : d8 143 : 7a 144 :
4c 145 : d4 146 : b8 147 : 54 148 : b7 149 : c3 150 : c9 151 : 7b 152 : a3 153 : 71 154 : 2b 155 :
b4 156 : 50 157 : 54 158 : 22 159 : 95 160 : df 161 : 17 162 : c9 163 : 41 164 : 80 165 : 2b 166 :
f0 167 : ba 168 : 4a 169 : a9 170 : 7f 171 : 13 172 : 1e 173 : 12 174 : a8 175 : 2 176 : 3 177 : 3d
178 : 71 179 : e6 180 : 96 181 : 89 182 : c6 183 : 46 184 : d6 185 : 1b 186 : 5f 187 : 20 188 : a0
189 : a3 190 : 49 191 : 79 192 : 9
193 : 53
-----
```

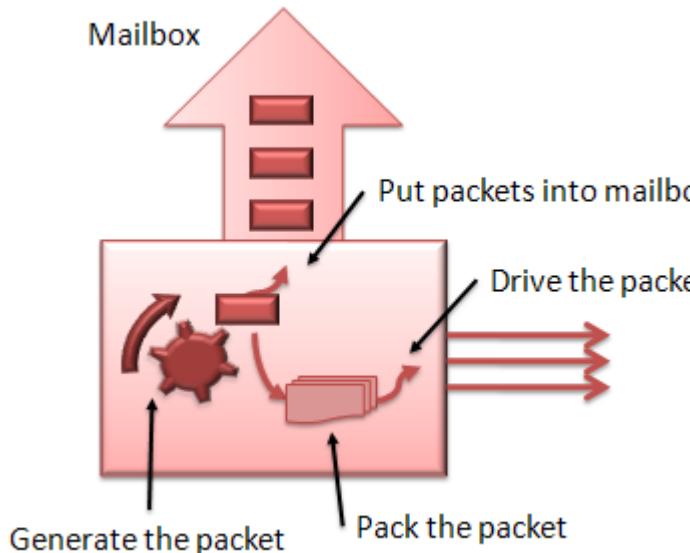
Packing,Unpacking and compare worked  
Randomization Sucessesfull.

.....  
.....  
.....

## **PHASE 5 DRIVER**

In phase 5 we will write a driver and then instantiate the driver in environment and send packet in to DUT. Driver class is defined in Driver.sv file.

Driver is class which generates the packets and then drives it to the DUT input interface and pushes the packet in to mailbox.



© www.testbench.in

- 1) Declare a packet.

```
packet gpkt;
```

- 2) Declare a virtual input\_interface of the switch. We will connect this to the Physical interface of the top module same as what we did in environment class.

```
virtual input_interface.IP input_intf;
```

- 3) Define a mailbox "drvr2sb" which is used to send the packets to the score board.

```
mailbox drvr2sb;
```

- 4) Define new constructor with arguments, virtual input interface and a mail box which is used to send packets from the driver to scoreboard.

```
function new(virtual input_interface.IP input_intf_new,mailbox drvr2sb);
  this.input_intf = input_intf_new ;
  if(drvr2sb == null)
    begin
```

```

$display(" **ERROR**: drvr2sb is null");
$finish;
end
else
this.drvr2sb = drvr2sb;

```

5) Construct the packet in the driver constructor.

```
gpkt = new();
```

6) Define the start method.

In start method, do the following

Repeat the following steps for num\_of\_pkts times.

```
repeat($root.num_of_pkts)
```

Randomize the packet and check if the randomization is successes full.

```

if ( pkt.randomize())
  begin
    $display (" %0d : Driver : Randomization Successes full.",$time);
    .....
    .....
  else
    begin
      $display (" %0d Driver : ** Randomization failed. **",$time);
    .....
    .....

```

Display the packet content.

```
pkt.display();
```

Then pack the packet in to bytes.

```
length = pkt.byte_pack(bytes);
```

Then send the packet byte in to the switch by asserting data\_status of the input interface signal and driving the data bytes on to the data\_in signal.

```

foreach(bytes[i])
begin
    @(posedge input_intf.clock);
    input_intf.cb.data_status <= 1;
    input_intf.cb.data_in <= bytes[i];
end

```

After driving all the data bytes, deassert data\_status signal of the input interface.

```

@(posedge input_intf.clock);
input_intf.cb.data_status <= 0;
input_intf.cb.data_in <= 0;

```

Send the packet in to mail "drvr2sb" box for scoreboard.

```
drvr2sb.put(pkt);
```

If randomization fails, increment the error counter which is defined in Globals.sv file

```
$root.error++;
```

#### Driver Class Source Code:

```

`ifndef GUARD_DRIVER
`define GUARD_DRIVER

class Driver;
    virtual input_interface.IP input_intf;
    mailbox drvr2sb;
    packet gpkt;
    //// constructor method ////
    function new(virtual input_interface.IP input_intf_new,mailbox drvr2sb);
        this.input_intf = input_intf_new ;
        if(drvr2sb == null)
            begin
                $display(" **ERROR**: drvr2sb is null");
                $finish;
            end
            else
                this.drvr2sb = drvr2sb;
                gpkt = new();
    endfunction : new

```

```

/// method to send the packet to DUT ///////
task start();
  packet pkt;
  int length;
  logic [7:0] bytes[];
  repeat($root.num_of_pkts)
  begin
    repeat(3) @(posedge input_intf.clock);
    pkt = new gpkt;
    /// Randomize the packet /////
    if ( pkt.randomize())
    begin
      $display (" %0d : Driver : Randomization Successes full. ",$time);
      /// display the packet content /////
      pkt.display();

    /// Pack the packet in tp stream of bytes /////
    length = pkt.byte_pack(bytes);

    /// assert the data_status signal and send the packed bytes /////
    foreach(bytes[i])
    begin
      @(posedge input_intf.clock);
      input_intf.cb.data_status <= 1;
      input_intf.cb.data_in <= bytes[i];
    end

    /// deassert the data_status singal /////
    @(posedge input_intf.clock);
    input_intf.cb.data_status <= 0;
    input_intf.cb.data_in <= 0;

    /// Push the packet in to mailbox for scoreboard /////
    drvr2sb.put(pkt);

    $display(" %0d : Driver : Finished Driving the packet with length %0d",$time,length);
  end
  else
  begin
    $display (" %0d Driver : ** Randomization failed. **",$time);
  
```

```

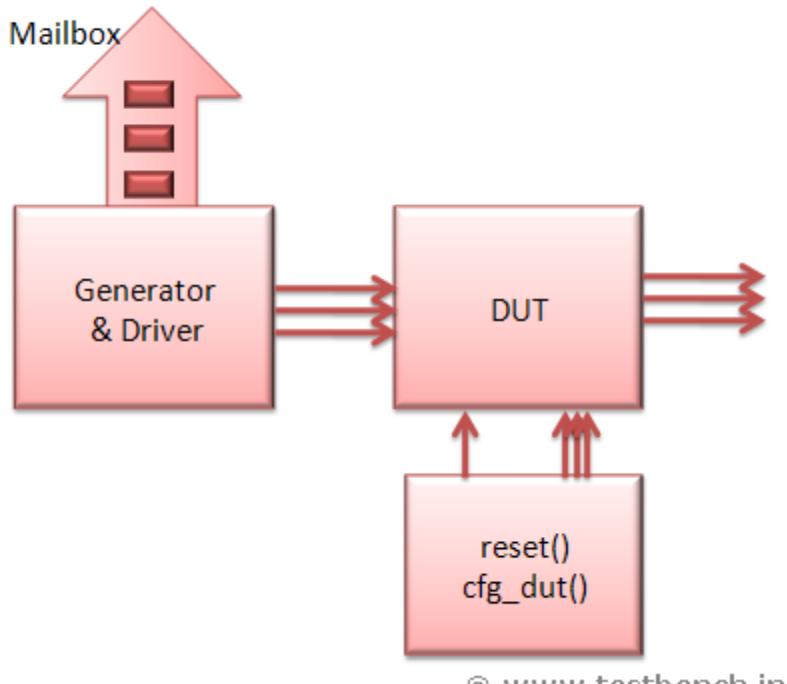
////// Increment the error count in randomization fails ///////
$root.error++;
end
end
endtask : start

endclass

```

`endif

Now we will take the instance of the driver in the environment class.



© www.testbench.in

1) Declare a mailbox "drvr2sb" which will be used to connect the scoreboard and driver.

**mailbox** drvr2sb;

2) Declare a driver object "drvr".

Driver drvr;

3) In build method, construct the mail box.

drvr2sb = new();

4) In build method, construct the driver object. Pass the input\_intf and "drvr2sb" mail box.

```
drvr= new(input_intf,drvr2sb);
```

5) To start sending the packets to the DUT, call the start method of "drvr" in the start method of Environment class.

```
drvr.start();
```

### **Environment Class Source Code:**

```
`ifndef GUARD_ENV
`define GUARD_ENV

class Environment ;
    virtual mem_interface.MEM    mem_intf      ;
    virtual input_interface.IP   input_intf    ;
    virtual output_interface.OP  output_intf[4] ;

    Driver drvr;
    mailbox drvr2sb;

function new(virtual mem_interface.MEM    mem_intf_new      ,
            virtual input_interface.IP   input_intf_new    ,
            virtual output_interface.OP  output_intf_new[4] );
    this.mem_intf      = mem_intf_new      ;
    this.input_intf    = input_intf_new    ;
    this.output_intf   = output_intf_new ;
    $display("%0d : Environment : created env object",$time);
endfunction : new

function void build();
    $display("%0d : Environment : start of build() method",$time);

    drvr2sb = new();
    drvr= new(input_intf,drvr2sb);

    $display("%0d : Environment : end of build() method",$time);
endfunction : build

task reset();
    $display("%0d : Environment : start of reset() method",$time);
    // Drive all DUT inputs to a known state
    mem_intf.cb.mem_data    <= 0;
    mem_intf.cb.mem_add     <= 0;
```

```

mem_intf.cb.mem_en      <= 0;
mem_intf.cb.mem_rd_wr   <= 0;
input_intf.cb.data_in    <= 0;
input_intf.cb.data_status <= 0;
output_intf[0].cb.read   <= 0;
output_intf[1].cb.read   <= 0;
output_intf[2].cb.read   <= 0;
output_intf[3].cb.read   <= 0;

// Reset the DUT
input_intf.reset     <= 1;
repeat (4) @ input_intf.clock;
input_intf.reset     <= 0;

$display("%0d : Environment : end of reset() method",$time);
endtask : reset

task cfg_dut();
$display("%0d : Environment : start of cfg_dut() method",$time);

mem_intf.cb.mem_en <= 1;
@(posedge mem_intf.clock);
mem_intf.cb.mem_rd_wr <= 1;

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h0;
mem_intf.cb.mem_data <= `P0;
$display("%0d : Environment : Port 0 Address %h",$time,`P0);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h1;
mem_intf.cb.mem_data <= `P1;
$display("%0d : Environment : Port 1 Address %h",$time,`P1);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h2;
mem_intf.cb.mem_data <= `P2;
$display("%0d : Environment : Port 2 Address %h",$time,`P2);

@(posedge mem_intf.clock);

```

```

mem_intf.cb.mem_add <= 8'h3;
mem_intf.cb.mem_data <= `P3;
$display(" %0d : Environment : Port 3 Address %h ",$time,`P3);

@(posedge mem_intf.clock);
mem_intf.cb.mem_en <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add <= 0;
mem_intf.cb.mem_data <= 0;

$display(" %0d : Environment : end of cfg_dut() method",$time);
endtask :cfg_dut

task start();
$display(" %0d : Environment : start of start() method",$time);

drvrv.start();

$display(" %0d : Environment : end of start() method",$time);
endtask : start

task wait_for_end();
$display(" %0d : Environment : start of wait_for_end() method",$time);
repeat(10000) @(input_intf.clock);
$display(" %0d : Environment : end of wait_for_end() method",$time);
endtask : wait_for_end

task run();
$display(" %0d : Environment : start of run() method",$time);
build();
reset();
cfg_dut();
start();
wait_for_end();
report();
$display(" %0d : Environment : end of run() method",$time);
endtask : run

task report();endtask : report

```

```
endclass  
`endif
```

### Download the phase 5 source code:

[switch\\_5.tar](#)

[Browse the code in switch\\_5.tar](#)

### Run the command:

```
vcs -sverilog -f filelist -R -ntb_opts dtm
```

### Log file report.

```
***** Start of testcase *****  
0 : Environment : created env object  
0 : Environment : start of run() method  
0 : Environment : start of build() method  
0 : Environment : end of build() method  
0 : Environment : start of reset() method  
40 : Environment : end of reset() method  
40 : Environment : start of cfg_dut() method  
70 : Environment : Port 0 Address 00  
90 : Environment : Port 1 Address 11  
110 : Environment : Port 2 Address 22  
130 : Environment : Port 3 Address 33  
150 : Environment : end of cfg_dut() method  
150 : Environment : start of start() method  
210 : Driver : Randomization Successes full.
```

----- PACKET KIND -----

```
fcs_kind    : BAD_FCS  
length_kind : GOOD_LENGTH
```

----- PACKET -----

```
0 : 22  
1 : 11  
2 : 2d  
3 : 63 4 : 2a 5 : 2e 6 : c 7 : a 8 : 14 9 : c1 10 : 14 11 : 8f 12 : 54 13 : 5d 14 : da 15 :  
22 16 : 2c 17 : ac 18 : 1c 19 : 48 20 : 3c 21 : 7e 22 : f3 23 : ed 24 : 24 25 : d1 26 : 3e 27 :  
38 28 : aa 29 : 54 30 : 19 31 : 89 32 : aa 33 : cf 34 : 67 35 : 19 36 : 9a 37 : 1d 38 : 96 39 :  
8 40 : 15 41 : 66 42 : 55 43 : b 44 : 70 45 : 35 46 : fc 47 : 8f  
48 : cd
```

1210 : Driver : Finished Driving the packet with length 49  
1270 : Driver : Randomization Successes full.

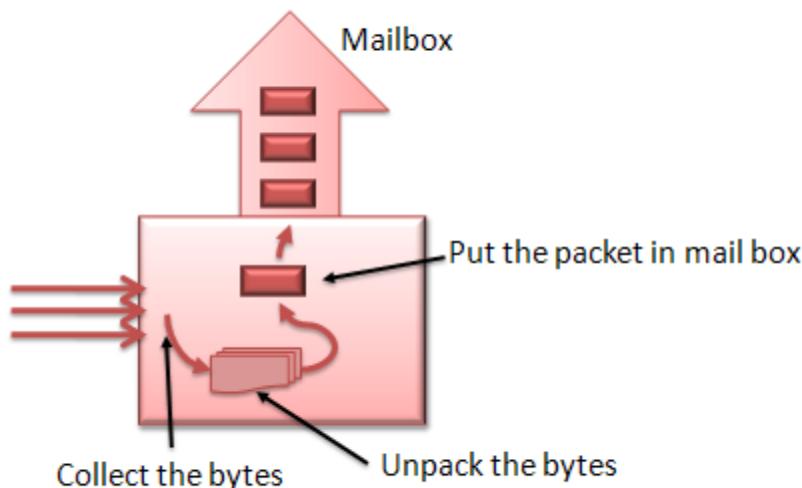
.....  
.....  
.....

## **PHASE 6 RECEIVER**

In this phase, we will write a receiver and use the receiver in environment class to collect the packets coming from the switch output\_interface.

Receiver collects the data bytes from the interface signal. And then unpacks the bytes in to packet and pushes it into mailbox.

Receiver class is written in receiver.sv file.



© www.testbench.in

1) Declare a virtual output\_interface. We will connect this to the Physical interface of the top module, same as what we did in environment class.

**virtual** output\_interface.OP output\_intf;

2) Declare a mailbox "rcvr2sb" which is used to send the packets to the score board

**mailbox** rcvr2sb;

3) Define new constructor with arguments, virtual input interface and a mail box which is used to send packets from the receiver to scoreboard.

**function new(virtual** output\_interface.OP output\_intf\_new,**mailbox** rcvr2sb);

```

this.output_intf = output_intf_new ;
if(rcvr2sb == null)
begin
    $display(" **ERROR**: rcvr2sb is null");
    $finish;
end
else
    this.rcvr2sb = rcvr2sb;
endfunction : new

```

4) Define the start method.

In start method, do the following

Wait for the ready signal to be asserted by the DUT.

```
wait(output_intf.cb.ready)
```

If the ready signal is asserted, then request the DUT to send the data out from the data\_out signal by asserting the read signal. When the data to be sent is finished by the DUT, it will deassert the ready signal. Once the ready signal is deasserted, stop collecting the data bytes and deassert the read signal.

```

output_intf.cb.read <= 1;
repeat(2) @(posedge output_intf.clock);
    while (output_intf.cb.ready)
        begin
            bytes = new[bytes.size + 1](bytes);
            bytes[bytes.size - 1] = output_intf.cb.data_out;
            @(posedge output_intf.clock);
        end
        output_intf.cb.read <= 0;
        @(posedge output_intf.clock);
        $display("%0d : Receiver : Received a packet of length %0d",$time,bytes.size);

```

Create a new packet object of packet.

```
pkt = new();
```

Then call the unpack method of the packet to unpack the bytes and then display the packet content.

```

pkt.byte_unpack(bytes);
pkt.display();

```

Then send the packet to scoreboard.

```
rcvr2sb.put(pkt);
```

Delete the dynamic array bytes.

```
bytes.delete();
```

### Receiver Class Source Code:

```

`ifndef GUARD_RECEIVER
`define GUARD_RECEIVER

class Receiver;
    virtual output_interface.OP output_intf;
    mailbox rcvr2sb;
    /// constructor method ///
    function new(virtual output_interface.OP output_intf_new,mailbox rcvr2sb);
        this.output_intf = output_intf_new ;
        if(rcvr2sb == null)
            begin
                $display(" **ERROR**: rcvr2sb is null");
                $finish;
            end
        else
            this.rcvr2sb = rcvr2sb;
    endfunction : new

    task start();
        logic [7:0] bytes[];
        packet pkt;
        forever
            begin
                repeat(2) @(posedge output_intf.clock);
                wait(output_intf.cb.ready)
                output_intf.cb.read <= 1;
                repeat(2) @(posedge output_intf.clock);
                while (output_intf.cb.ready)
                    begin
                        bytes = new[bytes.size + 1](bytes);
                        bytes[bytes.size - 1] = output_intf.cb.data_out;
                        @(posedge output_intf.clock);
                    end
                output_intf.cb.read <= 0;
                @(posedge output_intf.clock);
                $display("%0d : Receiver : Received a packet of length %0d",$time,bytes.size);
                pkt = new();
                pkt.byte_unpack(bytes);
                pkt.display();
                rcvr2sb.put(pkt);
                bytes.delete();
            end
    endtask
endclass

```

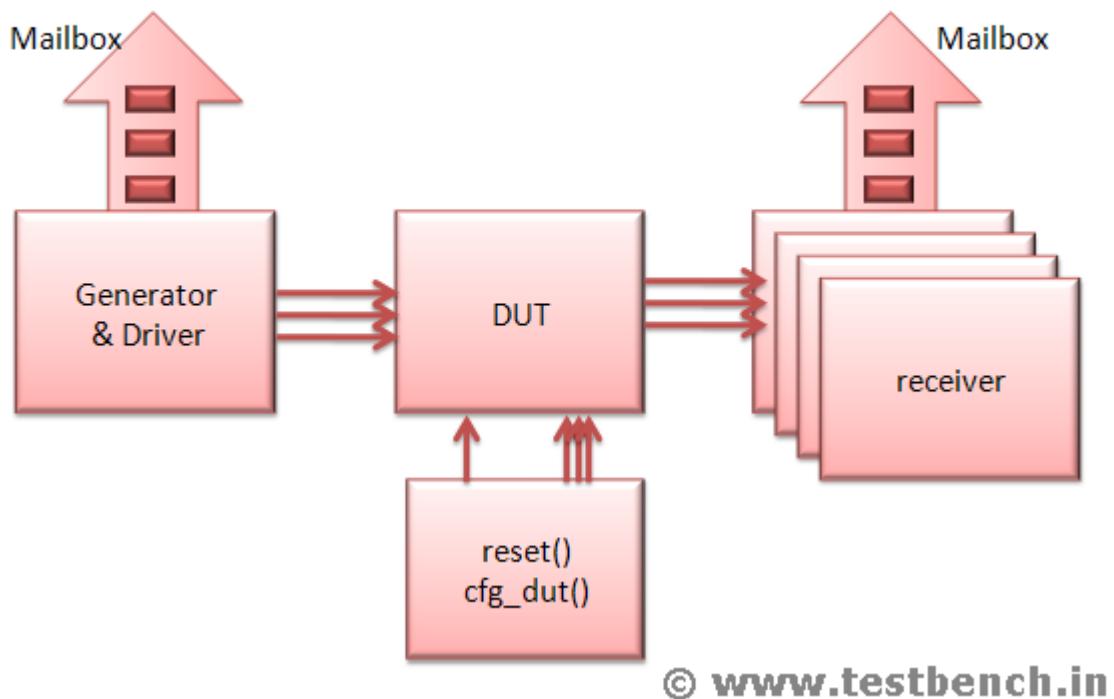
```

end
endtask : start

endclass
`endif

```

Now we will take the instance of the receiver in the environment class.



1) Declare a mailbox "rcvr2sb" which will be used to connect the scoreboard and receiver.

```
mailbox rcvr2sb;
```

2) Declare 4 receiver object "rcvr".

```
Receiver rcvr[4];
```

3) In build method, construct the mail box.

```
rcvr2sb = new();
```

4) In build method, construct the receiver object. Pass the output\_intf and "rcvr2sb" mail box. There are 4 output interfaces and receiver objects. We will connect one receiver for one output interface.

```
foreach(rcvr[i])
    rcvr[i]= new(output_intf[i],rcvr2sb);
```

5) To start collecting the packets from the DUT, call the "start" method of "rcvr" in the "start" method of Environment class.

```
task start();
    $display(" %0d : Environment : start of start() method",$time);
    fork
        drvr.start();
        rcvr[0].start();
        rcvr[1].start();
        rcvr[2].start();
        rcvr[3].start();
    join_any
    $display(" %0d : Environment : end of start() method",$time);
endtask : start
```

### Environment Class Source Code:

```
`ifndef GUARD_ENV
`define GUARD_ENV
class Environment ;
    virtual mem_interface.MEM    mem_intf      ;
    virtual input_interface.IP   input_intf    ;
    virtual output_interface.OP output_intf[4] ;
```

Driver drvr;

Receiver rcvr[4];

mailbox drvr2sb;

**mailbox** rcvr2sb;

```

function new(virtual mem_interface.MEM  mem_intf_new      ,
            virtual input_interface.IP  input_intf_new   ,
            virtual output_interface.OP output_intf_new[4] );

    this.mem_intf      = mem_intf_new  ;
    this.input_intf    = input_intf_new ;
    this.output_intf   = output_intf_new ;

    $display(" %0d : Environment : created env object",$time);
endfunction : new

function void build();
    $display(" %0d : Environment : start of build() method",$time);
    drvr2sb = new();

    rcvr2sb = new();  

    drvr= new(input_intf,drvr2sb);

    foreach(rcvr[i])
        rcvr[i]= new(output_intf[i],rcvr2sb);

    $display(" %0d : Environment : end of build() method",$time);
endfunction : build

task reset();
    $display(" %0d : Environment : start of reset() method",$time);
    // Drive all DUT inputs to a known state
    mem_intf.cb.mem_data      <= 0;
    mem_intf.cb.mem_add       <= 0;
    mem_intf.cb.mem_en        <= 0;
    mem_intf.cb.mem_rd_wr    <= 0;
    input_intf.cb.data_in     <= 0;
    input_intf.cb.data_status <= 0;
    output_intf[0].cb.read    <= 0;
    output_intf[1].cb.read    <= 0;
    output_intf[2].cb.read    <= 0;
    output_intf[3].cb.read    <= 0;

```

```

// Reset the DUT
input_intf.reset    <= 1;
repeat (4) @ input_intf.clock;
input_intf.reset    <= 0;

$display(" %0d : Environment : end of reset() method",$time);
endtask : reset

task cfg_dut();
$display(" %0d : Environment : start of cfg_dut() method",$time);

mem_intf.cb.mem_en <= 1;
@(posedge mem_intf.clock);
mem_intf.cb.mem_rd_wr <= 1;

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h0;
mem_intf.cb.mem_data <= `P0;
$display(" %0d : Environment : Port 0 Address %h ",$time,`P0);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h1;
mem_intf.cb.mem_data <= `P1;
$display(" %0d : Environment : Port 1 Address %h ",$time,`P1);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h2;
mem_intf.cb.mem_data <= `P2;
$display(" %0d : Environment : Port 2 Address %h ",$time,`P2);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h3;
mem_intf.cb.mem_data <= `P3;
$display(" %0d : Environment : Port 3 Address %h ",$time,`P3);

@(posedge mem_intf.clock);
mem_intf.cb.mem_en  <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add  <= 0;
mem_intf.cb.mem_data  <= 0;

```

```

$display(" %0d : Environment : end of cfg_dut() method",$time);
endtask :cfg_dut

task start();
$display(" %0d : Environment : start of start() method",$time);
fork
  drvr.start();

  rcvr[0].start();
  rcvr[1].start();
  rcvr[2].start();
  rcvr[3].start();

join_any
$display(" %0d : Environment : end of start() method",$time);
endtask : start

task wait_for_end();
$display(" %0d : Environment : start of wait_for_end() method",$time);
repeat(10000) @(input_intf.clock);
$display(" %0d : Environment : end of wait_for_end() method",$time);
endtask : wait_for_end

task run();
$display(" %0d : Environment : start of run() method",$time);
build();
reset();
cfg_dut();
start();
wait_for_end();
report();
$display(" %0d : Environment : end of run() method",$time);

endtask : run

task report();
endtask: report
endclass

```

```
`endif
```

[Download the phase 6 source code:](#)

[switch\\_6.tar](#)

[Browse the code in switch\\_6.tar](#)

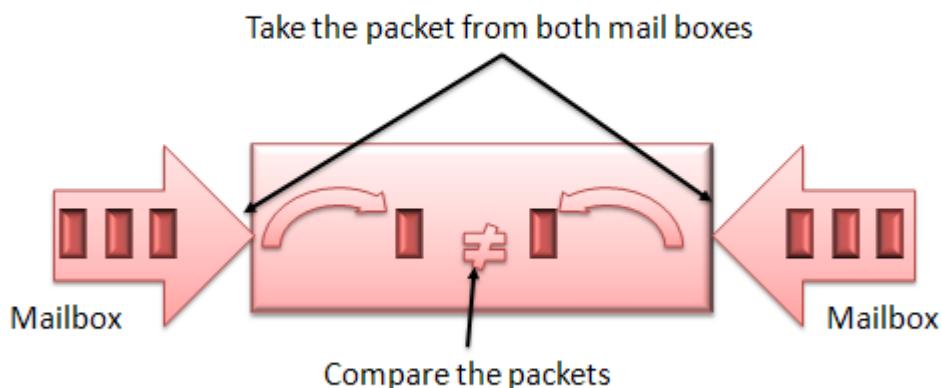
[Run the command:](#)

```
vcs -sverilog -f filelist -R -ntb_opts dtm
```

## **PHASE 7 SCOREBOARD**

In this phase we will see the scoreboard implementation.

Scoreboard has 2 mailboxes. One is used to for getting the packets from the driver and other from the receiver. Then the packets are compared and if they don't match, then error is asserted. Scoreboard is implemented in file Scoreboard.sv.



© [www.testbench.in](http://www.testbench.in)

- 1) Declare 2 mailboxes drvr2sb and rcvr2sb.

```
mailbox drvr2sb;
```

```
mailbox rcvr2sb;
```

- 2) Declare a constructor method with "drvr2sb" and "rcvr2sb" mailboxes as arguments.

```
function new(mailbox drvr2sb,mailbox rcvr2sb);
```

- 3) Connect the mailboxes of the constructor to the mail boxes of the scoreboard.

```
this.drvr2sb = drvr2sb;  
this.rcvr2sb = rcvr2sb;
```

4) Define a start method.

Do the following steps forever.

Wait until there is a packet in "rcvr2sb". Then pop the packet from the mail box.

```
rcvr2sb.get(pkt_rcv);  
$display("%0d : Scoreboard received a packet from receiver ", $time);
```

Then pop the packet from drvr2sb.

```
drvr2sb.get(pkt_exp);
```

Compare both packets and increment an error counter if they are not equal.

```
if(pkt_rcv.compare(pkt_exp))  
$display("%0d : Scoreboardd :Packet Matched ", $time);  
else  
$root.error++;
```

### Scoreboard Class Source Code:

```
`ifndef GUARD_SCOREBOARD  
`define GUARD_SCOREBOARD  
  
class Scoreboard;  
  
mailbox drvr2sb;  
mailbox rcvr2sb;  
  
function new(mailbox drvr2sb, mailbox rcvr2sb);  
this.drvr2sb = drvr2sb;  
this.rcvr2sb = rcvr2sb;  
endfunction:  
  
  
task start();  
packet pkt_rcv, pkt_exp;  
forever
```

```

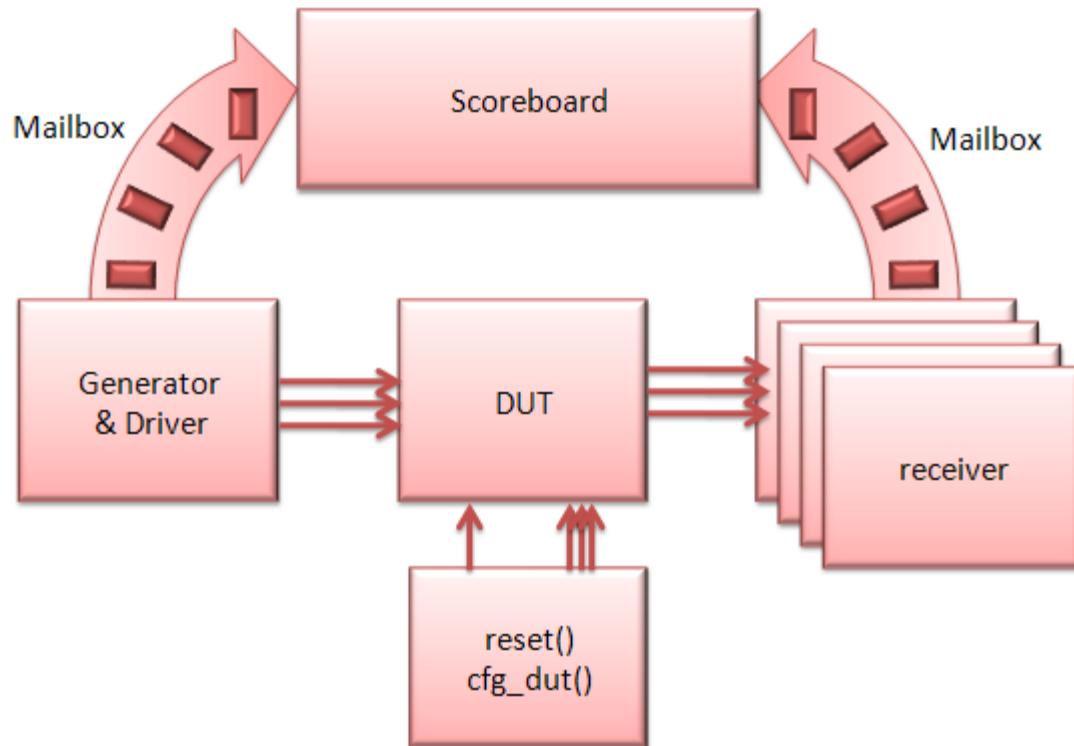
begin
  rcvr2sb.get(pkt_rcv);
  $display("%0d : Scoreboard : Scoreboard received a packet from receiver ",$time);
  drvr2sb.get(pkt_exp);
  if(pkt_rcv.compare(pkt_exp))
    $display("%0d : Scoreboard : Packet Matched ",$time);
  else
    $root.error++;
end
endtask : start

endclass

`endif

```

Now we will see how to connect the scoreboard in the Environment class.



1) Declare a scoreboard.

```
Scoreboard sb;
```

2) Construct the scoreboard in the build method. Pass the drvr2sb and rcvr2sb mailboxes to the score board constructor.

```
sb = new(drvr2sb,rcvr2sb);
```

3) Start the scoreboard method in the start method.

```
sb.start();
```

4) Now we are to the end of building the verification environment.

In the report() method of environment class, print the TEST PASS or TEST FAIL status based on the error count.

```
task report();
$display("\n\n*****");
if( 0 == $root.error)
    $display("*****      TEST PASSED      *****");
else
    $display("*****  TEST Failed with %0d errors *****",$root.error);

$display("*****\n");
endtask : report
```

### Source Code Of The Environment Class:

```
`ifndef GUARD_ENV
`define GUARD_ENV
class Environment ;
    virtual mem_interface.MEM    mem_intf    ;
    virtual input_interface.IP   input_intf   ;
    virtual output_interface.OP  output_intf[4] ;

    Driver drvr;
    Receiver rcvr[4];
```

```

Scoreboard sb;

mailbox drvr2sb ;
mailbox rcvr2sb ;

function new(virtual mem_interface.MEM mem_intf_new ,
            virtual input_interface.IP input_intf_new ,
            virtual output_interface.OP output_intf_new[4] );
    this.mem_intf = mem_intf_new ;
    this.input_intf = input_intf_new ;
    this.output_intf = output_intf_new ;
    $display("%0d : Environment : created env object",$time);
endfunction : new

function void build();
    $display("%0d : Environment : start of build() method",$time);
    drvr2sb = new();
    rcvr2sb = new();

    sb = new(drvr2sb,rcvr2sb);

    drvr= new(input_intf,drvr2sb);
    foreach(rcvr[i])
        rcvr[i]= new(output_intf[i],rcvr2sb);
    $display("%0d : Environment : end of build() method",$time);
endfunction : build

task reset();
    $display("%0d : Environment : start of reset() method",$time);
    // Drive all DUT inputs to a known state
    mem_intf.cb.mem_data <= 0;
    mem_intf.cb.mem_add <= 0;
    mem_intf.cb.mem_en <= 0;
    mem_intf.cb.mem_rd_wr <= 0;
    input_intf.cb.data_in <= 0;
    input_intf.cb.data_status <= 0;
    output_intf[0].cb.read <= 0;

```

```

output_intf[1].cb.read  <= 0;
output_intf[2].cb.read  <= 0;
output_intf[3].cb.read  <= 0;

// Reset the DUT
input_intf.reset      <= 1;
repeat (4) @ input_intf.clock;
input_intf.reset      <= 0;

$display(" %0d : Environment : end of reset() method",$time);
endtask : reset

task cfg_dut();
$display(" %0d : Environment : start of cfg_dut() method",$time);

mem_intf.cb.mem_en <= 1;
@(posedge mem_intf.clock);
mem_intf.cb.mem_rd_wr <= 1;

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h0;
mem_intf.cb.mem_data <= `P0;
$display(" %0d : Environment : Port 0 Address %h ",$time,`P0);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h1;
mem_intf.cb.mem_data <= `P1;
$display(" %0d : Environment : Port 1 Address %h ",$time,`P1);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h2;
mem_intf.cb.mem_data <= `P2;
$display(" %0d : Environment : Port 2 Address %h ",$time,`P2);

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h3;
mem_intf.cb.mem_data <= `P3;
$display(" %0d : Environment : Port 3 Address %h ",$time,`P3);

@(posedge mem_intf.clock);

```

```

mem_intf.cb.mem_en  <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add  <= 0;
mem_intf.cb.mem_data <= 0;

$display(" %0d : Environment : end of cfg_dut() method",$time);
endtask :cfg_dut

task start();
    $display(" %0d : Environment : start of start() method",$time);
    fork
        drvr.start();
        rcvr[0].start();
        rcvr[1].start();
        rcvr[2].start();
        rcvr[3].start();

        sb.start(); // This line is highlighted in blue

    join_any
    $display(" %0d : Environment : end of start() method",$time);
endtask : start

task wait_for_end();
    $display(" %0d : Environment : start of wait_for_end() method",$time);
    repeat(10000) @(input_intf.clock);
    $display(" %0d : Environment : end of wait_for_end() method",$time);
endtask : wait_for_end

task run();
    $display(" %0d : Environment : start of run() method",$time);
    build();
    reset();
    cfg_dut();
    start();
    wait_for_end();
    report();
    $display(" %0d : Environment : end of run() method",$time);
endtask: run

```

**Download the phase 7 score code:**

## switch\_7.tar

Browse the code in switch 7.tar

## Run the simulation:

```
vcs -sverilog -f filelist -R -ntb_opts dtm
```

## **PHASE 8 COVERAGE**

In this phase we will write the functional coverage for switch protocol. Functional coverage is written in Coverage.sv file. After running simulation, you will analyze the coverage results and find out if some test scenarios have not been exercised and write tests to exercise them.

The points which we need to cover are

- 1) Cover all the port address configurations.
  - 2) Cover all the packet lengths.
  - 3) Cover all correct and incorrect length fields.
  - 4) Cover good and bad FCS.
  - 5) Cover all the above combinations.

- 1) Define a cover group with following cover points.

- a) All packet lengths:

length : **coverpoint** pkt.length;

b) All port address:

```
da : coverpoint pkt.da {  
    bins p0 = { `P0 };  
    bins p1 = { `P1 };  
    bins p2 = { `P2 };  
    bins p3 = { `P3 }; }
```

c) Correct and incorrect Length field types:

```
length_kind : coverpoint pkt.length_kind;
```

d) Good and Bad FCS:

```
fcs_kind : coverpoint pkt.fcs_kind;
```

5) Cross product of all the above cover points:

```
all_cross: cross length,da,length_kind,fcs_kind;
```

2) In constructor method, construct the cover group

```
function new();  
    switch_coverage = new();  
endfunction : new
```

3) Write task which calls the sample method to cover the points.

```
task sample(packet pkt);  
    this(pkt) = pkt;  
    switch_coverage.sample();  
endtask:sample
```

### Source Code Of Coverage Class:

```
`ifndef GUARD_COVERAGE  
`define GUARD_COVERAGE
```

```

class coverage;
packet pkt;

covergroup switch_coverage;

length : coverpoint pkt.length;
da   : coverpoint pkt.da {
    bins p0 = { `P0 };
    bins p1 = { `P1 };
    bins p2 = { `P2 };
    bins p3 = { `P3 };
length_kind : coverpoint pkt.length_kind;
fcs_kind : coverpoint pkt.fcs_kind;

all_cross: cross length,da,length_kind,fcs_kind;
endgroup

function new();
switch_coverage = new();
endfunction : new

task sample(packet pkt);
this.pkt = pkt;
switch_coverage.sample();
endtask:sample

endclass

```

`endif

Now we will use this coverage class instance in scoreboard.

1) Take an instance of coverage class and construct it in scoreboard class.

coverage cov = new();

2) Call the sample method and pass the exp\_pkt to the sample method.

cov.sample(pkt\_exp);

### Source Code Of The Scoreboard Class:

```

`ifndef GUARD_SCOREBOARD
`define GUARD_SCOREBOARD

```

```

class Scoreboard;

```

```

mailbox drvr2sb;
mailbox rcvr2sb;

coverage cov = new();

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

endclass

```

`endif

**Download the phase 8 score code:**

[switch\\_8.tar](#)

[Browse the code in switch\\_8.tar](#)

**Run the simulation:**

```
vcs -sverilog -f filelist -R -ntb_opts dtm
urg -dir simv.cm
```

## **PHASE 9 TESTCASE**

In this phase we will write a constraint random testcase.

Lets verify the DUT by sending large packets of length above 200.

1) In testcase file, define a small\_packet class.

This calls is inherited from the packet class and data.size() field is constraint to generate the packet with size greater than 200.

```
class small_packet extends packet;
constraint small_c { data.size > 200 ; }
endclass
```

2) In program block, create an object of the small\_packet class.

Then call the build method of env.

```
small_packet spkt;
```

3) Pass the object of the small\_packet to the packet handle which is in driver.

```
env.drvr.gpkt = spkt;
```

Then call the reset(),cfg\_dut(),start(),wait\_for\_end() and report() methods as in the run method.

```
env.reset();
env.cfg_dut();
env.start();
env.wait_for_end();
env.report();
```

### **Source Code Of Constraint Testcase:**

```
'ifndef GUARD_TESTCASE
`define GUARD_TESTCASE
class small_packet extends packet;
constraint small_c { data.size > 200 ; }
endclass
program testcase(mem_interface.MEM mem_intf,input_interface.IP input_intf,output_interface.OP output_intf[4]);
Environment env;
small_packet spkt;
initial
begin
$display(" ***** Start of testcase *****");
spkt = new();
env = new(mem_intf,input_intf,output_intf);
env.build();
env.drvr.gpkt = spkt;
env.reset();
env.cfg_dut();
env.start();
```

```
env.wait_for_end();
env.report();
#1000;
end
final
$display(" ***** End of testcase *****");
endprogram
`endif
```

**Download the phase 9 source code:**

[switch\\_9.tar](#)

[Browse the code in switch\\_9.tar](#)

**Run the simulation:**

```
vcs -sverilog -f filelist -R -ntb_opts dtm
urg -dir simv.cm
```

## **INDEX**

### **INTRODUCTION**

Installing Uvm Library

### **SPECIFICATION**

- Switch Specification
- Packet Format
- Configuration
- Interface Specification

### **VERIFICATION PLAN**

- Overview
- Feature Extraction
- Stimulus Generation Plan
- Verification Environment

### **PHASE 1 TOP**

- Interface
- Top Module

### **PHASE 2 CONFIGURATION**

- Configuration
- Updates To Top Module

### **PHASE 3 ENVIRONMENT N TESTCASE**

- Environment
- Testcase

### **PHASE 4 PACKET**

- Packet
- Test The Transaction Implementation

### **PHASE 5 SEQUENCER N SEQUENCE**

- Sequencer
- Sequence

### **PHASE 6 DRIVER**

- Driver
- Environment Updates
- Testcase Updates

### **PHASE 7 RECEIVER**

- Receiver
- Environment Class Updates

### **PHASE 8 SCOREBOARD**

- Scoreboard
- Environment Class Updates

## **INTRODUCTION**

In this tutorial, we will verify the Switch RTL core using UVM in SystemVerilog. Following are the steps we follow to verify the Switch RTL core.

- 1) Understand the specification
- 2) Developing Verification Plan
- 3) Building the Verification Environment. We will build the Environment in Multiple phases, so it will be easy for you to learn step by step.

In this verification environment, I will not use agents and monitors to make this tutorial simple and easy.

Phase 1) We will develop the interfaces, and connect it to DUT in top module.

Phase 2) We will develop the Configuration class.

Phase 3) We will develop the Environment class and Simple testcase and simulate them.

Phase 4) We will develop packet class based on the stimulus plan. We will also write a small code to test the packet class implementation.

Phase 5) We will develop sequencer and a sample sequences.

Phase 6) We will develop driver and connect it to the Sequencer in to environment.

Phase 7) We will develop receiver and instantiate in environment.

Phase 8) We will develop scoreboard which does the comparison of the expected packet with the actual packet received from the DUT and connect it to driver and receiver in Environment class.

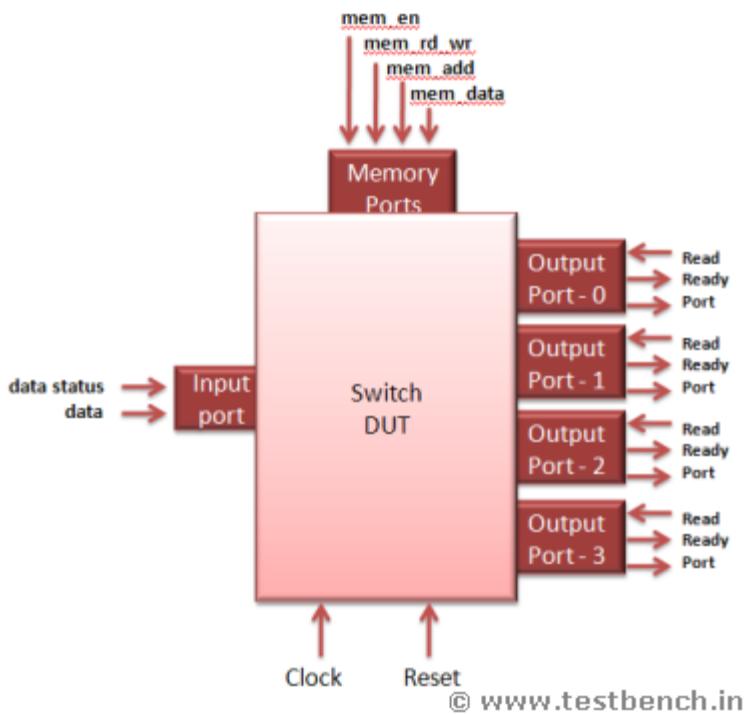
## **Installing Uvm Library**

- 1) Go to <http://www.accellera.org/activities/vip/>
- 2) Download the uvm\*.tar.gz file.
- 3) Untar the file.
- 4) Go to the extracted directory : cd uvm\*\uvm\src
- 5) Set the UVM\_HOME path : setenv UVM\_HOME `pwd`  
(This is required to run the examples which are downloaded from this site)
- 6) Go to examples : cd ..\examples\hello\_world\uvm\
- 7) Compile the example using :  
your\_tool\_compilation\_command -f compile\_<toolname>.f  
(example for questasim use : qverilog -f compile\_questa.f)
- 8) Run the example.

## **SPECIFICATION**

### **Switch Specification:**

This is a simple switch. Switch is a packet based protocol. Switch drives the incoming packet which comes from the input port to output ports based on the address contained in the packet. The switch has a one input port from which the packet enters. It has four output ports where the packet is driven out.



### **Packet Format:**

Packet contains 3 parts. They are Header, data and frame check sequence.

Packet width is 8 bits and the length of the packet can be between 4 bytes to 259 bytes.

#### **Packet header:**

Packet header contains three fields DA, SA and length.

② DA: Destination address of the packet is of 8 bits. The switch drives the packet to respective ports based on this destination address of the packets. Each output port has 8-bit unique port address. If the destination address of the packet matches the port address, then switch drives the packet to the output port.

② SA: Source address of the packet from where it originate. It is 8 bits.

② Length: Length of the data is of 8 bits and from 0 to 255. Length is measured in terms of bytes.

If Length = 0, it means data length is 0 bytes

If Length = 1, it means data length is 1 bytes

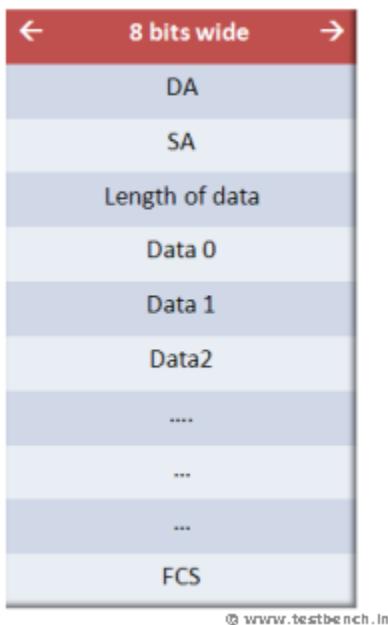
If Length = 2, it means data length is 2 bytes

If Length = 255, it means data length is 255 bytes

☞ Data: Data should be in terms of bytes and can take anything.

☞ FCS: Frame check sequence

This field contains the security check of the packet. It is calculated over the header and data.



### Configuration:

Switch has four output ports. These output ports address have to be configured to a unique address. Switch matches the DA field of the packet with this configured port address and sends the packet on to that port. Switch contains a memory. This memory has 4 locations, each can store 8 bits. To configure the switch port address, memory write operation has to be done using memory interface. Memory address (0,1,2,3) contains the address of port(0,1,2,3) respectively.

### Interface Specification:

The Switch has one input Interface, from where the packet enters and 4 output interfaces from where the packet comes out and one memory interface, through the port address can be configured. Switch also has a clock and asynchronous reset signal.

## MEMORY INTERFACE:

Through memory interfaced output port address are configured. It accepts 8 bit data to be written to memory. It has 8 bit address inputs. Address 0,1,2,3 contains the address of the port 0,1,2,3 respectively.

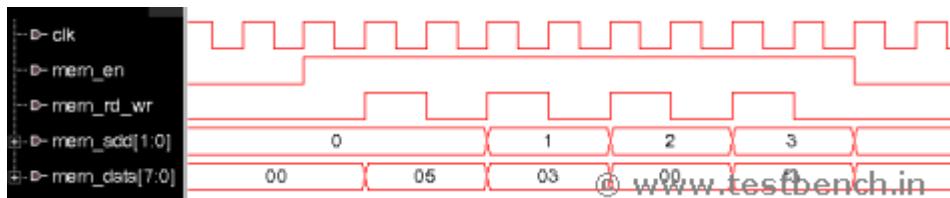
There are 4 input signals to memory interface. They are

```
input mem_en;  
input mem_rd_wr;  
input [1:0] mem_add;  
input [7:0] mem_data;
```

All the signals are active high and are synchronous to the positive edge of clock signal.

To configure a port address,

1. Assert the mem\_en signal.
2. Asser the mem\_rd\_wr signal.
3. Drive the port number (0 or 1 or 2 or 3) on the mem\_add signal
4. Drive the 8 bit port address on to mem\_data signal.



## INPUT PORT

Packets are sent into the switch using input port.

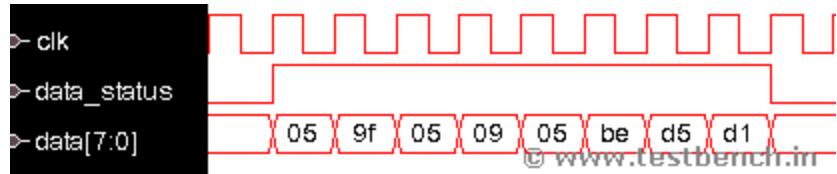
All the signals are active high and are synchronous to the positive edge of clock signal.

**input** port has **2 input** signals. They are

```
input [7:0] data;  
input data_status;
```

To send the packet in to switch,

1. Assert the data\_status signal.
2. Send the packet on the data signal byte by byte.
3. After sending all the data bytes, deassert the data\_status signal.
4. There should be at least 3 clock cycles difference between packets.



## OUTPUT PORT

Switch sends the packets out using the output ports. There are 4 ports, each having data, ready and read signals. All the signals are active high and are synchronous to the positive edge of clock signal.

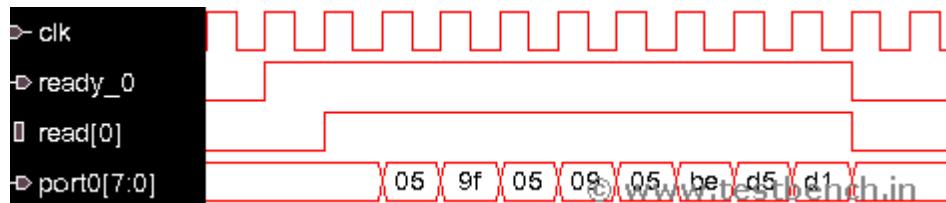
Signal list is

```

output [7:0] port0;
output [7:0] port1;
output [7:0] port2;
output [7:0] port3;
output ready_0;
output ready_1;
output ready_2;
output ready_3;
input read_0;
input read_1;
input read_2;
input read_3;
```

When the data is ready to be sent out from the port, switch asserts ready\_\* signal high indicating that data is ready to be sent.

If the read\_\* signal is asserted, when ready\_\* is high, then the data comes out of the port\_\* signal after one clock cycle.



## RTL code:

RTL code is attached with the tar files. From the Phase 1, you can download the tar files.

## **VERIFICATION PLAN**

### **Overview**

This Document describes the Verification Plan for Switch. The Verification Plan is based on System Verilog Hardware Verification Language. The methodology used for Verification is Constraint random coverage driven verification.

### **Feature Extraction**

This section contains list of all the features to be verified.

1)ID: Configuration

Description: Configure all the 4 port address with unique values.

2)ID: Packet DA

Description: DA field of packet should be any of the port address. All the 4 port address should be used.

3) ID : Packet payload

Description: Length can be from 1 to 255. Send packets with all the lengths.

4) ID: Length

Description:

Length field contains length of the payload.

5) ID: FCS

Description:

Good FCS: Send packet with good FCS.

Bad FCS: Send packet with corrupted FCS.

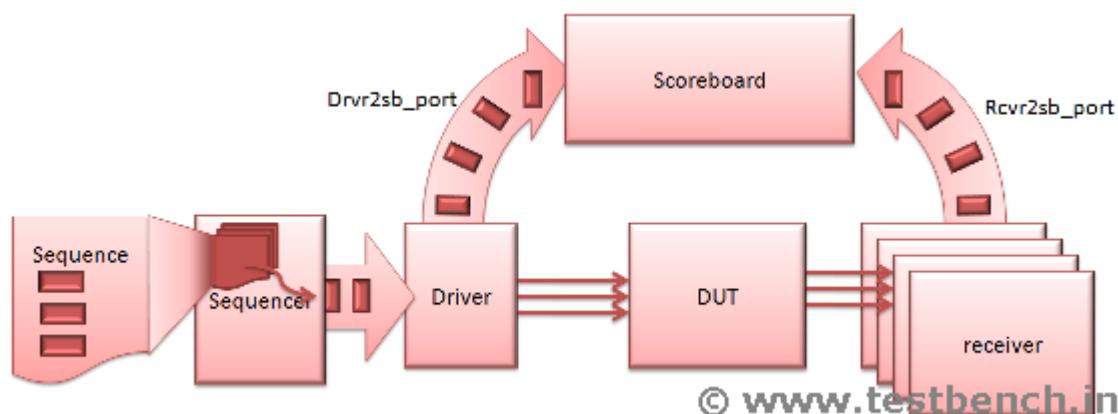
### **Stimulus Generation Plan**

1) Packet DA: Generate packet DA with the configured address.

2) Payload length: generate payload length ranging from 2 to 255.

3) Generate good and bad FCS.

### **Verification Environment**



## PHASE 1 TOP

In phase 1,

- 1) We will write SystemVerilog Interfaces for input port, output port and memory port.
- 2) We will write Top module where testcase and DUT instances are done.
- 3) DUT and interfaces are connected in top module.
- 4) We will implement Clock generator in top module.

## Interface

In the interface.sv file, declare the 3 interfaces in the following way.

All the interfaces has clock as input.

All the signals in interface are wire type.

All the signals are synchronized to clock except reset in clocking block.

This approach will avoid race conditions between the design and the verification environment.

Define the set-up and hold time using parameters.

Signal directional w.r.t TestBench is specified with modport.

## Interface Source Code

```
'ifndef GUARD_INTERFACE
`define GUARD_INTERFACE
///////////////////////////////
// Interface declaration for the memory///
///////////////////////////////

interface mem_interface(input bit clock);
    parameter setup_time = 5ns;
    parameter hold_time = 3ns;

    wire [7:0] mem_data;
    wire [1:0] mem_add;
    wire      mem_en;
    wire      mem_rd_wr;

    clocking cb@(posedge clock);
        default input #setup_time output #hold_time;
        output   mem_data;
        output   mem_add;
        output   mem_en;
        output   mem_rd_wr;
    endclocking:cb

    modport MEM(clocking cb,input clock);

endinterface :mem_interface
```

```

///////////
// Interface for the input side of switch.//
// Reset signal is also passed here. //
///////////

interface input_interface(input bit clock);

parameter setup_time = 5ns;
parameter hold_time = 3ns;

wire      data_status;
wire      [7:0] data_in;
reg       reset;

clocking cb@(posedge clock);
  default input #setup_time output #hold_time;
  output   data_status;
  output   data_in;
endclocking:cb

modport IP(clocking cb,output reset,input clock);

endinterface:input_interface

///////////
// Interface for the output side of the switch.//
// output_interface is for only one output port//
///////////

interface output_interface(input bit clock);

parameter setup_time = 5ns;
parameter hold_time = 3ns;

wire      [7:0] data_out;
wire      ready;
wire      read;

clocking cb@(posedge clock);
  default input #setup_time output #hold_time;
  input    data_out;

```

```

input ready;
output read;
endclocking:cb

modport OP(clocking cb,input clock);

endinterface:output_interface

//////////`endif

```

### **Top Module**

The modules that are included in the source text but are not instantiated are called top modules. This module is the highest scope of modules. Generally this module is named as "top" and referenced as "top module". Module name can be anything. This top-level module will contain the design portion of the simulation.  
Do the following in the top module:

- 1) The first step is to import the uvm packages

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

- 2) Generate the clock signal.

```
bit Clock;

initial
  begin
    #20;
    forever #10 Clock = ~Clock;
  end
```

- 2) Do the instances of memory interface.

```
mem_interface mem_intf(Clock);
```

- 3) Do the instances of input interface.

```
input_interface input_intf(Clock);
```

4) There are 4 output ports. So do 4 instances of output\_interface.

```
output_interface output_intf[4](Clock);
```

5) Connect all the interfaces and DUT. The design which we have taken is in verilog. So Verilog DUT instance is connected signal by signal.

```
switch DUT (.clk(Clock),
    .reset(input_intf.reset),
    .data_status(input_intf.data_status),
    .data(input_intf.data_in),
    .port0(output_intf[0].data_out),
    .port1(output_intf[1].data_out),
    .port2(output_intf[2].data_out),
    .port3(output_intf[3].data_out),
    .ready_0(output_intf[0].ready),
    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),
    .mem_en(mem_intf.mem_en),
    .mem_rd_wr(mem_intf.mem_rd_wr),
    .mem_add(mem_intf.mem_add),
    .mem_data(mem_intf.mem_data));
```

### Top module Scource Code

```
`ifndef GUARD_TOP
#define GUARD_TOP
///////////////////////////////
// Importing UVM Packages          //
///////////////////////////////
`include "uvm.svh"
import uvm_pkg::*;


```

```

module top();

// Clock Declaration and Generation          //
///////////////////////////////////////////////////////////////////
bit Clock;

initial
begin
    #20;
    forever #10 Clock = ~Clock;
end
///////////////////////////////////////////////////////////////////
// Memory interface instance          //
///////////////////////////////////////////////////////////////////

mem_interface mem_intf(Clock);

///////////////////////////////////////////////////////////////////
// Input interface instance          //
///////////////////////////////////////////////////////////////////

input_interface input_intf(Clock);

///////////////////////////////////////////////////////////////////
// output interface instance          //
///////////////////////////////////////////////////////////////////

output_interface output_intf[4](Clock);

///////////////////////////////////////////////////////////////////
// DUT instance and signal connection      //
///////////////////////////////////////////////////////////////////


switch DUT (.clk(Clock),
    .reset(input_intf.reset),
    .data_status(input_intf.data_status),
    .data(input_intf.data_in),
    .port0(output_intf[0].data_out),

```

```

    .port1(output_intf[1].data_out),
    .port2(output_intf[2].data_out),
    .port3(output_intf[3].data_out),
    .ready_0(output_intf[0].ready),
    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),
    .mem_en(mem_intf.mem_en),
    .mem_rd_wr(mem_intf.mem_rd_wr),
    .mem_add(mem_intf.mem_add),
    .mem_data(mem_intf.mem_data));

```

**endmodule : top**

`endif

[Download the files:](#)

[uvm\\_switch\\_1.tar](#)

[Browse the code in uvm\\_switch\\_1.tar](#)

[Command to compile](#)

VCS Users : make vcs

Questa Users: make questa

## **PHASE 2 CONFIGURATION**

In this phase we will implement the configuration class. All the requirements of the testbench configurations will be declared inside this class. Virtual interfaces required by verification components driver and receiver for connecting to DUT are declared in this class. We will also declare 4 variables which will hold the port address of the DUT.

uvm\_object does not have the simulation phases and can be used in get\_config\_object and set\_config\_object method. So we will implement the configuration class by extending uvm\_object.

[Configuration](#)

- 1) Define configuration class by extending uvm\_object

```

`ifndef GUARD_CONFIGURATION
`define GUARD_CONFIGURATION

class Configuration extends uvm_object;
endclass : Configuration

```

`endif

2) Declare All the interfaces which are required in this verification environment.

```

virtual input_interface.IP  input_intf;
virtual mem_interface.MEM  mem_intf;
virtual output_interface.OP output_intf[4];

```

3) Declare 4 variables which holds the device port address.

```
bit [7:0] device_add[4];
```

4) uvm\_object required to define the uvm\_object::creat() method.

uvm\_object::create method allocates a new object of the same type as this object and returns it via a base uvm\_object handle.

In create method, we have to construct a new object of configuration class and update all the important fields and return it.

```

virtual function uvm_object create(string name="");
Configuration t = new();

```

```

t.device_add = this.device_add;
t.input_intf = this.input_intf;
t.mem_intf   = this.mem_intf;
t.output_intf= this.output_intf;

```

```
return t;
```

```
endfunction : create
```

### Configuration class source code

```

`ifndef GUARD_CONFIGURATION
`define GUARD_CONFIGURATION

```

```
class Configuration extends uvm_object;
```

```

virtual input_interface.IP  input_intf;
virtual mem_interface.MEM  mem_intf;

```

```
virtual output_interface.OP output_intf[4];
bit [7:0] device_add[4];
virtual function uvm_object create(string name = "");
    Configuration t = new();
    t.device_add = this.device_add;
    t.input_intf = this.input_intf;
    t.mem_intf = this.mem_intf;
    t.output_intf = this.output_intf;
    return t;
endfunction : create

endclass : Configuration
```

In top module we will crea

In top module we will crea

In top module we will create an object of the above defined configuration class and update the interfaces so that all the verification components can access to physical interfaces in top module using configuration class object.

- 1) Declare a Configuration class object  
Configuration cfg;
  - 2) Construct the configuration object and update the interfaces.

## initial begin

```
cfg = new();  
cfg.input_intf = input_intf;  
cfg.mem_intf = mem_intf;  
cfg.output_intf = output_intf;
```

- 3) In top module , we have to call the run\_test() method.

```
run_test();
```

## Top module updates

```
typedef class Configuration;
module top();
// Clock Declaration and Generation //
bit Clock;
initial
begin
#20;
forever #10 Clock = ~Clock;
end
//
```

```

// Memory interface instance          //
///////////////////////////////
mem_interface mem_intf(Clock);
///////////////////////////////

// Input interface instance          //
///////////////////////////////
input_interface input_intf(Clock);
///////////////////////////////

// output interface instance         //
///////////////////////////////
output_interface output_intf[4](Clock);

///////////////////////////////
// Creat Configuration and Strart the run_test//
///////////////////////////////

Configuration cfg;
initial begin
  cfg = new();
  cfg.input_intf = input_intf;
  cfg.mem_intf = mem_intf;
  cfg.output_intf = output_intf;

  run_test();
end
///////////////////////////////
// DUT instance and signal connection      //
///////////////////////////////

switch DUT (.clk(Clock),
  .reset(input_intf.reset),
  .data_status(input_intf.data_status),
  .data(input_intf.data_in),
  .port0(output_intf[0].data_out),
  .port1(output_intf[1].data_out),
  .port2(output_intf[2].data_out),
  .port3(output_intf[3].data_out),
  .ready_0(output_intf[0].ready),
  .ready_1(output_intf[1].ready),
  .ready_2(output_intf[2].ready),
  .ready_3(output_intf[3].ready),
  .read_0(output_intf[0].read),

```

```

.read_1(output_intf[1].read),
.read_2(output_intf[2].read),
.read_3(output_intf[3].read),
.mem_en(mem_intf.mem_en),
.mem_rd_wr(mem_intf.mem_rd_wr),
.mem_add(mem_intf.mem_add),
.mem_data(mem_intf.mem_data));
endmodule : top
`endif

```

### Download the source code

[uvm\\_switch\\_2.tar](#)

[Browse the code in uvm\\_switch\\_2.tar](#)

### Command to compile

VCS Users : make vcs

Questa Users: make questa

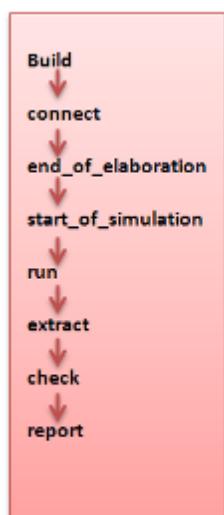
## **PHASE 3 ENVIRONMENT N TESTCASE**

In the phase we will implement the skeleton for environment class.

We will declare virtual interfaces and Extend Required Environment class virtual methods.

We will also implement a simple testcase and run the simulation.

Environment class is used to implement verification environments in UVM. It is extension on uvm\_env class. The testbench simulation needs some systematic flow like building the components, connection the components, starting the components etc. uvm\_env base class has methods formalize the simulation steps. All methods are declared as virtual methods.



We will not implement all the uvm\_env virtual methods in this phase but will we print messages from these methods which are required for this example to understand the simulation execution. Testcase contains the instance of the environment class. This testcase Creates a Environment object and defines the required test specific functionality.

Verification environment contains the declarations of the virtual interfaces. These virtual interfaces are pointed to the physical interfaces which are declared in the top module. These virtual interfaces are made to point to physical interface in the testcase.

## **Environment**

- 1) Extend uvm\_env class to define Environment class.

```
`ifndef GUARD_ENV
`define GUARD_ENV
class Environment extends uvm_env;
endclass : Environment
`endif
```

- 2) Declare the utility macro. This utility macro provides the implementation of create() and get\_type\_name() methods.

```
`uvm_component_utils(Environment)
```

- 3) Define the constructor. In the constructor, call the super methods and pass the parent object. Parent is the object in which environment is instantiated.

```
function new(string name , uvm_component parent = null);
    super.new(name, parent);
endfunction: new
```

- 4) Define build method. In build method, just print messages and super.build() must be called. This method is automatically called.

Build is the first phase in simulation. This phase is used to construct the child components of the current class.

```
virtual function void build();
    super.build();
    uvm_report_info(get_full_name(),"START of build ",UVM_LOW);
    uvm_report_info(get_full_name(),"END of build ",UVM_LOW);
endfunction
```

- 5) Define connect method. In connect method, just print messages and super.connect() must be called.

This method is called automatically after the build() method is called. This method is used for connecting port and exports.

```
virtual function void connect();
    super.connect();
    uvm_report_info(get_full_name(),"START of connect ",UVM_LOW);
    uvm_report_info(get_full_name(),"END of connect ",UVM_LOW);
```

**endfunction**

**Environment class Source Code**

```
`ifndef GUARD_ENV
`define GUARD_ENV
class Environment extends uvm_env;

    `uvm_component_utils(Environment)
function new(string name , uvm_component parent = null);
    super.new(name, parent);
endfunction: new
virtual function void build();
    super.build();
    uvm_report_info(get_full_name(),"START of build ",UVM_LOW);
    uvm_report_info(get_full_name(),"END of build ",UVM_LOW);
endfunction
virtual function void connect();
    super.connect();
    uvm_report_info(get_full_name(),"START of connect ",UVM_LOW);

    uvm_report_info(get_full_name(),"END of connect ",UVM_LOW);
endfunction
```

**endclass : Environment**

**`endif**

**Testcase**

Now we will implement testcase. In UVM, testcases are implemented by extending uvm\_test. Using uvm\_test , provides the ability to select which test to execute using the UVM\_TESTNAME command line option or argument to the uvm\_root::run\_test task. We will use UVM\_TESTNAME command line argument.

1) Define a testcase by extending uvm\_test class.

```
class test1 extends uvm_test;
endclass
```

2) Declare the utility macro.

```
`uvm_component_utils(test1)
```

3) Take the instance of Environemtn.

```
Environment t_env ;
```

4) Define the constructor method.

In this method, construct the environment class object and dont forget to pass the parent argument.

```

function new (string name="test1", uvm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new

```

5) Define run() method.

run() method is the only task which is time consuming. After completing the start\_of\_simulation() phase , this method is called.

To terminate this task, we will use global\_stop\_request().

As we dont have anything now to write in this testcase, just call the global\_stop\_request() after some delay.

```

virtual task run ();
    #3000ns;
    global_stop_request();
endtask : run

```

With this, for the first time, we can do the simulation.

#### Testcase Source code

```

class test1 extends uvm_test;

```

```

`uvm_component_utils(test1)

```

```

Environment t_env ;

```

```

function new (string name="test1", uvm_component parent=null);

```

```

    super.new (name, parent);

```

```

    t_env = new("t_env",this);

```

```

endfunction : new

```

```

virtual task run ();

```

```

    #3000ns;

```

```

    global_stop_request();

```

```

endtask : run

```

```

endclass : test1

```

## Download the Source Code

[uvm\\_switch\\_3.tar](#)

[Browse the code in uvm\\_switch\\_3.tar](#)

## Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

## Log report after simulation

```
UVM_INFO @ 0 [RNTST] Running test test1...
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] START of build
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] END of build
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] START of connect
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] END of connect
```

--- UVM Report Summary ---

```
** Report counts by severity
UVM_INFO : 5
UVM_WARNING : 0
UVM_ERROR : 0
UVM_FATAL : 0
** Report counts by id
[RNTST ] 1
[uvm_test_top.t_env ] 4
```

## **PHASE 4 PACKET**

In this Phase, we will develop Transaction as per the verification plan. We will define required methods and constraints. We will also develop a small logic to test our implementation of this class.

### **Packet**

We will write the packet class in Packet.sv file. Packet class variables and constraints have been derived from stimulus generation plan.

One way to model Packet is by extending uvm\_sequence\_item. uvm\_sequence\_item provides basic functionality for sequence items and sequences to operate in a sequence mechanism. Packet class should be able to generate all possible packet types randomly. To define copy, compare,

record, print and sprint methods, we will use UVM field macros. For packing and Unpacking, we will define the logic and not use the field macros.

#### Revisit Stimulus Generation Plan

- 1) Packet DA: Generate packet DA with the configured address.
- 2) Payload length: generate payload length ranging from 2 to 255.
- 3) Generate good and bad FCS.

1) Define enumerated type data for fcs.

```
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;
```

2) Define transaction by extending uvm\_sequence\_item.

```
class Packet extends uvm_sequence_item;  
endclass : Packet
```

3) Define all the fields as rand variables.

```
rand fcs_kind_t fcs_kind;
```

```
rand bit [7:0] length;  
rand bit [7:0] da;  
rand bit [7:0] sa;  
rand bit [7:0] data[];  
rand byte fcs;
```

4) Define constraints to constraint payload size of data.

```
constraint payload_size_c { data.size inside { [2 : 255]; } }  
constraint length_c { length == data.size; }
```

5) Define the constructor method.

```
function new(string name = "");  
    super.new(name);  
endfunction : new
```

6) In post\_randomize() , define the fcs value based on fcs\_kind.

```
function void post_randomize();  
    if(fcs_kind == GOOD_FCS)  
        fcs = 8'b0;  
    else  
        fcs = 8'b1;  
    fcs = cal_fcs();  
endfunction : post_randomize
```

7) Define cal\_fcs() method which computes the fcs value.

```
//// method to calculate the fcs ////  
virtual function byte cal_fcs;  
    return da ^ sa ^ length ^ data.xor() ^ fcs;
```

```
endfunction : cal_fcs
```

- 8) Using uvm\_field\_\* macros, define transaction required method.

We will define packing and unpacking methods manually, so use UVM\_NOPACK for excluding atomic creation of packing and un packing method.

```
`uvm_object_utils_begin(Packet)
`uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(sa, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(length, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_array_int(data, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(fcs, UVM_ALL_ON|UVM_NOPACK)
`uvm_object_utils_end
```

- 9) Define do\_pack() method which does the packing operation.

```
function void do_pack(uvm_packer packer);
    super.do_pack(packer);
    packer.pack_field_int(da,$bits(da));
    packer.pack_field_int(sa,$bits(sa));
    packer.pack_field_int(length,$bits(length));
    foreach(data[i])
        packer.pack_field_int(data[i],8);
    packer.pack_field_int(fcs,$bits(fcs));
endfunction : do_pack
```

- 10) Define do\_unpack() method which does the unpacking operation.

```
function void do_unpack(uvm_packer packer);
    int sz;
    super.do_pack(packer);

    da = packer.unpack_field_int($bits(da));
    sa = packer.unpack_field_int($bits(sa));
    length = packer.unpack_field_int($bits(length));

    data.delete();
    data = new[length];
    foreach(data[i])
        data[i] = packer.unpack_field_int(8);
    fcs = packer.unpack_field_int($bits(fcs));
endfunction : do_unpack
```

#### Packet class source code

```
`ifndef GUARD_PACKET
`define GUARD_PACKET
```

```

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

//Define the enumerated types for packet types
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;

class Packet extends uvm_sequence_item;

    rand fcs_kind_t   fcs_kind;
    rand bit [7:0] length;
    rand bit [7:0] da;
    rand bit [7:0] sa;
    rand bit [7:0] data[];
    rand byte fcs;

    constraint payload_size_c { data.size inside { [1 : 6]};}
    constraint length_c { length == data.size; }

    function new(string name = "");
        super.new(name);
    endfunction : new

    function void post_randomize();
        if(fcs_kind == GOOD_FCS)
            fcs = 8'b0;
        else
            fcs = 8'b1;
        fcs = cal_fcs();
    endfunction : post_randomize

    //// method to calculate the fcs /////
    virtual function byte cal_fcs;
        return da ^ sa ^ length ^ data.xor() ^ fcs;
    endfunction : cal_fcs

`uvm_object_utils_begin(Packet)
`uvm_field_int(da, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(sa, UVM_ALL_ON|UVM_NOPACK)

```

```

`uvm_field_int(length, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_array_int(data, UVM_ALL_ON|UVM_NOPACK)
`uvm_field_int(fcs, UVM_ALL_ON|UVM_NOPACK)
`uvm_object_utils_end

function void do_pack(uvm_packer packer);
  super.do_pack(packer);
  packer.pack_field_int(da,$bits(da));
  packer.pack_field_int(sa,$bits(sa));
  packer.pack_field_int(length,$bits(length));
  foreach(data[i])
    packer.pack_field_int(data[i],8);
  packer.pack_field_int(fcs,$bits(fcs));
endfunction : do_pack

function void do_unpack(uvm_packer packer);
  int sz;
  super.do_pack(packer);

  da = packer.unpack_field_int($bits(da));
  sa = packer.unpack_field_int($bits(sa));
  length = packer.unpack_field_int($bits(length));

  data.delete();
  data = new[length];
  foreach(data[i])
    data[i] = packer.unpack_field_int(8);
  fcs = packer.unpack_field_int($bits(fcs));
endfunction : do_unpack

```

**endclass** : Packet

### Test The Transaction Implementation

Now we will write a small logic to test our packet implantation. This module is not used in normal verification.

Define a module and take the instance of packet class. Randomize the packet and call the print method to analyze the generation. Then pack the packet in to bytes and then unpack bytes and then call compare method to check all the method implementation.

1) Declare Packet objects and dynamic arrays.

```
Packet pkt1 = new("pkt1");
Packet pkt2 = new("pkt2");
byte unsigned pkdbytes[];
```

2) In a initial block, randomize the packet, pack the packet in to pkdbytes and then unpack it and compare the packets.

```
if(pkt1.randomize)
begin
    $display(" Randomization Sucessesfull.");
    pkt1.print();
    uvm_default_packer.use_metadata = 1;
    void'(pkt1.pack_bytes(pkdbytes));
    $display("Size of pkd bits %d",pkdbytes.size());
    pkt2.unpack_bytes(pkdbytes);
    pkt2.print();
    if(pkt2.compare(pkt1))
        $display(" Packing,Unpacking and compare worked");
    else
        $display(" *** Something went wrong in Packing or Unpacking or compare *** \n \n");
```

#### Logic to test the transaction implementation

```
module test;
```

```
Packet pkt1 = new("pkt1");
Packet pkt2 = new("pkt2");
byte unsigned pkdbytes[];

initial
repeat(10)
    if(pkt1.randomize)
begin
    $display(" Randomization Successesfull.");
    pkt1.print();
    uvm_default_packer.use_metadata = 1;
    void'(pkt1.pack_bytes(pkdbytes));
    $display("Size of pkd bits %d",pkdbytes.size());
    pkt2.unpack_bytes(pkdbytes);
```

```

pkt2.print();
if(pkt2.compare(pkt1))
    $display(" Packing,Unpacking and compare worked");
else
    $display(" *** Something went wrong in Packing or Unpacking or compare *** \n \n");
end
else
    $display(" *** Randomization Failed ***");

```

**endmodule**

[Download the Source Code](#)

[uvm\\_switch\\_4.tar](#)

[Browse the code in uvm\\_switch\\_4.tar](#)

[Command to run the simulation](#)

VCS Users : make vcs

Questa Users: make questa

[Log report after simulation](#)

Randomization Sucessesfull.

Name	Type	Size	Value
pkt1	Packet	-	pkt1@3
--da	integral	8	'ha5
--sa	integral	8	'ha1
--length	integral	8	'h6
--data	da(integral)	6	-
----[0]	integral	8	'h58
----[1]	integral	8	'h60
----[2]	integral	8	'h34
----[3]	integral	8	'hdd
----[4]	integral	8	'h9
----[5]	integral	8	'haf
--fcs	integral	8	'h75

Size of pkd bits      10

Name	Type	Size	Value
pkt2	Packet	-	pkt2@5
--da	integral	8	'ha5
--sa	integral	8	'ha1
--length	integral	8	'h6
--data	da(integral)	6	-
---[0]	integral	8	'h58
---[1]	integral	8	'h60
---[2]	integral	8	'h34
---[3]	integral	8	'hdd
---[4]	integral	8	'h9
---[5]	integral	8	'haf
--fcs	integral	8	'h75

Packing,Unpacking and compare worked

....

....

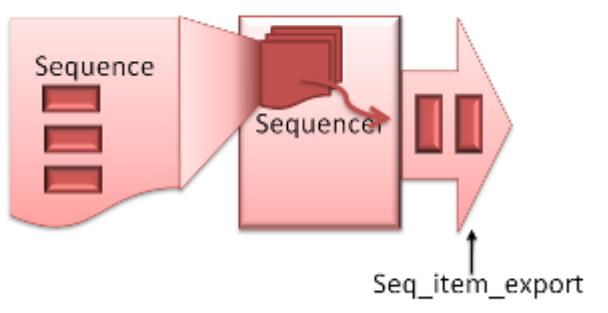
....

## .....PHASE 5 SEQUENCER N SEQUENCE

In this phase we will develop Sequence and Sequencer.

A sequence is series of transaction and sequencer is used to for controlling the flow of transaction generation.

A sequence of transaction (which we already developed in previous phase) is defined by extending uvm\_sequence class. uvm\_sequencer does the generation of this sequence of transaction, uvm\_driver takes the transaction from Sequencer and processes the packet/ drives to other component or to DUT.



## Sequencer

A Sequencer is defined by extending uvm\_sequencer. uvm\_sequencer has a port seq\_item\_export which is used to connect to uvm\_driver for transaction transfer.

1) Define a sequencer by extending uvm\_sequence.

```
`ifndef GUARD_SEQUENCER
`define GUARD_SEQUENCER

class Sequencer extends uvm_sequencer #(Packet);
```

```
endclass : Sequencer
```

```
`endif
```

2) We need Device port address, which are in configuration class. So declare a configuration class object.

```
Configuration cfg;
```

3) Declare Sequencer utility macros.

```
`uvm_sequencer_utils(Sequencer)
```

4) Define the constructor.

```
function new (string name, uvm_component parent);
    super.new(name, parent);
    `uvm_update_sequence_lib_and_item(Packet)
endfunction : new
```

5) In end\_of\_elaboration() method, using get\_config\_object(), get the configuration object which will be passed from testcase.

get\_config\_object() returns object of type uvm\_object, so using a temporary uvm\_object and cast it to configuration object.

```
virtual function void end_of_elaboration();
    uvm_object tmp;
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
endfunction
```

### Sequencer source code

```
`ifndef GUARD_SEQUENCER
`define GUARD_SEQUENCER

class Sequencer extends uvm_sequencer #(Packet);
Configuration cfg;
`uvm_sequencer_utils(Sequencer)
function new (string name, uvm_component parent);
super.new(name, parent);
`uvm_update_sequence_lib_and_item(Packet)
endfunction : new
virtual function void end_of_elaboration();
uvm_object tmp;
assert(get_config_object("Configuration",tmp));
$cast(cfg,tmp);
endfunction
endclass : Sequencer
`endif
```

### Sequence

A sequence is defined by extending uvm\_sequence class. This sequence of transactions should be defined in body() method of uvm\_sequence class. UVM has macros and methods to define the transaction types. We will use macros in this example.

You can define as many sequences as you want. We will define 2 sequences.

1) Define sequence by extending

```
class Seq_device0_and_device1 extends uvm_sequence #(Packet);
endclass: Seq_device0_and_device1
```

2) Define constructor method.

```
function new(string name = "Seq_do");
super.new(name);
endfunction : new
```

3) Declare utilities macro. With this macro, this sequence is tied to Sequencer.

```
`uvm_sequence_utils(Seq_device0_and_device1, Sequencer)
```

4) The algorithm for the transaction should be defined in body() method of the sequence. In this sequence we will define the algorithm such that alternate transactions for device port 0 and 1 are generated.

The device addresses are available in configuration object which is in sequencer. Every sequence has a handle to its sequence through p\_sequencer. Using p\_sequencer handle, access the device address.

```
virtual task body();
forever begin
```

```

`uvm_do_with(item, {da == p_sequencer.cfg.device_add[0];} );
`uvm_do_with(item, {da == p_sequencer.cfg.device_add[1];} );
end
endtask : body

```

### Sequence Source Code

```

class Seq_device0_and_device1 extends uvm_sequence #(Packet);
function new(string name = "Seq_device0_and_device1");
    super.new(name);
endfunction : new
Packet item;
`uvm_sequence_utils(Seq_device0_and_device1, Sequencer)
virtual task body();
    forever begin
        `uvm_do_with(item, {da == p_sequencer.cfg.device_add[0];} );
        `uvm_do_with(item, {da == p_sequencer.cfg.device_add[1];} );
    end
endtask : body

```

**endclass :Seq\_device0\_and\_device1**

### One more Sequence

```

class Seq_constant_length extends uvm_sequence #(Packet);
function new(string name = "Seq_constant_length");
    super.new(name);
endfunction : new
Packet item;
`uvm_sequence_utils(Seq_constant_length, Sequencer)
virtual task body();
    forever begin
        `uvm_do_with(item, {length == 10;da == p_sequencer.cfg.device_add[0];} );
    end
endtask : body
endclass : Seq_constant_length

```

### Download the Source Code

[uvm\\_switch\\_5.tar](#)

[Browse the code in uvm\\_switch\\_5.tar](#)

### Command to run the simulation

VCS Users : make vcs

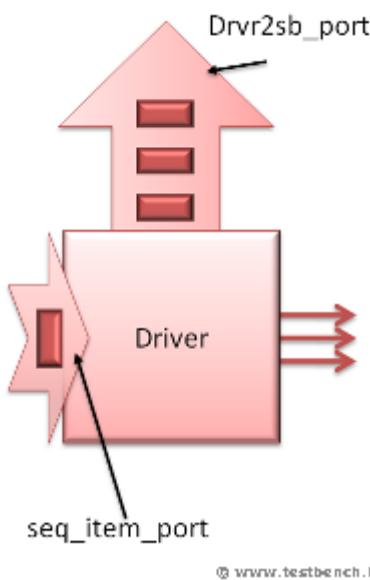
Questa Users: make questa

## **PHASE 6 DRIVER**

### **Driver**

In this phase we will develop the driver. Driver is defined by extending uvm\_driver. Driver takes the transaction from the sequencer using seq\_item\_port. This transaction will be driven to DUT as per the interface specification. After driving the transaction to DUT, it sends the transaction to scoreboard using uvm\_analysis\_port.

In driver class, we will also define task for resetting DUT and configuring the DUT. After completing the driver class implementation, we will instantiate it in environment class and connect the sequencer to it. We will also update the test case and run the simulation to check the implementation which we did till now.



- 1) Define the driver class by extending uvm\_driver;

```
'ifndef GUARD_DRIVER
`define GUARD_DRIVER
class Driver extends uvm_driver #(Packet);
endclass : Driver
```

- 2) Create a handle to configuration object. Using this object we can get DUT interfaces and DUT port addresses.

```
Configuration cfg;
```

3) Declare input and memory interfaces

```
virtual input_interface.IP input_intf;  
virtual mem_interface.MEM mem_intf;
```

4) Declare uvm\_analysis\_port which is used to send packets to scoreboard.

```
uvm_analysis_port #(Packet) Drvr2Sb_port;
```

5) Declare component utilities macro.

```
`uvm_component_utils(Driver)
```

6) Define the constructor method. Pass the parent object to super class.

```
function new( string name = "", uvm_component parent = null );  
    super.new( name , parent );  
endfunction : new
```

7) In the build method and construct Drvr2Sb\_port object.

```
virtual function void build();  
    super.build();  
    Drvr2Sb_port = new("Drvr2Sb_port", this);  
endfunction : build
```

8) In the end\_of\_elaboration() method, get the configuration object using get\_config\_object and update the virtual interfaces.

```
virtual function void end_of_elaboration();  
    uvm_object tmp;  
    super.end_of_elaboration();  
    assert(get_config_object("Configuration",tmp));  
    $cast(cfg,tmp);  
    this.input_intf = cfg.input_intf;  
    this.mem_intf = cfg.mem_intf;  
endfunction : end_of_elaboration
```

9) Define the reset\_dut() method which will be used for resetting the DUT.

```
virtual task reset_dut();  
    uvm_report_info(get_full_name(),"Start of reset_dut() method ",UVM_LOW);  
    mem_intf.mem_data    <= 0;  
    mem_intf.mem_add    <= 0;  
    mem_intf.mem_en     <= 0;  
    mem_intf.mem_rd_wr  <= 0;  
    input_intf.data_in   <= 0;  
    input_intf.data_status <= 0;
```

```

input_intf.reset    <= 1;
repeat (4) @ input_intf.clock;
input_intf.reset    <= 0;

uvm_report_info(get_full_name(),"End of reset_dut() method ",UVM_LOW);
endtask : reset_dut

```

10) Define the cfg\_dut() method which does the configuration due port address.

```

virtual task cfg_dut();
  uvm_report_info(get_full_name(),"Start of cfg_dut() method ",UVM_LOW);
  mem_intf.cb.mem_en <= 1;
  @(mem_intf.cb);
  mem_intf.cb.mem_rd_wr <= 1;

  foreach (cfg.device_add[i]) begin

    @(mem_intf.cb);
    mem_intf.cb.mem_add <= i;
    mem_intf.cb.mem_data <= cfg.device_add[i];
    uvm_report_info(get_full_name(),$psprintf(" Port %0d Address %h
",i,cfg.device_add[i]),UVM_LOW);

  end

  @(mem_intf.cb);
  mem_intf.cb.mem_en <= 0;
  mem_intf.cb.mem_rd_wr <= 0;
  mem_intf.cb.mem_add <= 0;
  mem_intf.cb.mem_data <= 0;

  uvm_report_info(get_full_name(),"End of cfg_dut() method ",UVM_LOW);
endtask : cfg_dut

```

11) Define drive() method which will be used to drive the packet to DUT. In this method pack the packet fields using the pack\_bytes() method of the transaction and drive the packed data to DUT interface.

```

virtual task drive(Packet pkt);
    byte unsigned bytes[];
    int pkt_len;
    pkt_len = pkt.pack_bytes(bytes);
    uvm_report_info(get_full_name(),"Driving packet ...",UVM_LOW);

    foreach(bytes[i])
    begin
        @(input_intf.cb);
        input_intf.data_status <= 1 ;
        input_intf.data_in <= bytes[i];
    end

    @(input_intf.cb);
    input_intf.data_status <= 0 ;
    input_intf.data_in <= 0;
    repeat(2) @(input_intf.cb);
endtask : drive

```

12) Now we will use the above 3 defined methods and update the run() method of uvm\_driver. First call the reset\_dut() method and then cfg\_dut(). After completing the configuration, in a forever loop get the transaction from seq\_item\_port and send it DUT using drive() method and also to scoreboard using Drvr2SB\_port .

```

virtual task run();
    Packet pkt;
    @(input_intf.cb);
    reset_dut();
    cfg_dut();
    forever begin
        seq_item_port.get_next_item(pkt);
        Drvr2Sb_port.write(pkt);
        @(input_intf.cb);
        drive(pkt);
        @(input_intf.cb);
        seq_item_port.item_done();
    end
endtask : run

```

## Driver class source code

```
`ifndef GUARD_DRIVER
`define GUARD_DRIVER

class Driver extends uvm_driver #(Packet);

Configuration cfg;

virtual input_interface.IP  input_intf;
virtual mem_interface.MEM  mem_intf;

uvm_analysis_port #(Packet) Drvr2Sb_port;

`uvm_component_utils(Driver)

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

virtual function void build();
    super.build();
    Drvr2Sb_port = new("Drvr2Sb", this);
endfunction : build

virtual function void end_of_elaboration();
    uvm_object tmp;
    super.end_of_elaboration();
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
    this.input_intf = cfg.input_intf;
    this.mem_intf = cfg.mem_intf;
endfunction : end_of_elaboration

virtual task run();
    Packet pkt;
    @(input_intf.cb);
    reset_dut();
    cfg_dut();
    forever begin
```

```

seq_item_port.get_next_item(pkt);
DrvR2Sb_port.write(pkt);
@(input_intf.cb);
drive(pkt);
@(input_intf.cb);
seq_item_port.item_done();
end
endtask : run

virtual task reset_dut();
  uvm_report_info(get_full_name(),"Start of reset_dut() method ",UVM_LOW);
  mem_intf.mem_data    <= 0;
  mem_intf.mem_add     <= 0;
  mem_intf.mem_en      <= 0;
  mem_intf.mem_rd_wr   <= 0;
  input_intf.data_in   <= 0;
  input_intf.data_status <= 0;

  input_intf.reset     <= 1;
  repeat (4) @ input_intf.clock;
    input_intf.reset     <= 0;

  uvm_report_info(get_full_name(),"End of reset_dut() method ",UVM_LOW);
endtask : reset_dut

virtual task cfg_dut();
  uvm_report_info(get_full_name(),"Start of cfg_dut() method ",UVM_LOW);
  mem_intf.cb.mem_en <= 1;
  @(mem_intf.cb);
  mem_intf.cb.mem_rd_wr <= 1;

  foreach (cfg.device_add[i]) begin

    @(mem_intf.cb);
    mem_intf.cb.mem_add <= i;
    mem_intf.cb.mem_data <= cfg.device_add[i];
    uvm_report_info(get_full_name(),$psprintf(" Port %0d Address %h
",i,cfg.device_add[i]),UVM_LOW);

  end

```

```

@(mem_intf.cb);
mem_intf.cb.mem_en  <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add  <= 0;
mem_intf.cb.mem_data <= 0;

uvm_report_info(get_full_name(),"End of cfg_dut() method ",UVM_LOW);
endtask : cfg_dut

virtual task drive(Packet pkt);
    byte unsigned bytes[];
    int pkt_len;
    pkt_len = pkt.pack_bytes(bytes);
    uvm_report_info(get_full_name(),"Driving packet ...",UVM_LOW);

    foreach(bytes[i])
        begin
            @(input_intf.cb);
            input_intf.data_status <= 1 ;
            input_intf.data_in <= bytes[i];
        end

        @(input_intf.cb);
        input_intf.data_status <= 0 ;
        input_intf.data_in <= 0;
        repeat(2) @(input_intf.cb);
    endtask : drive

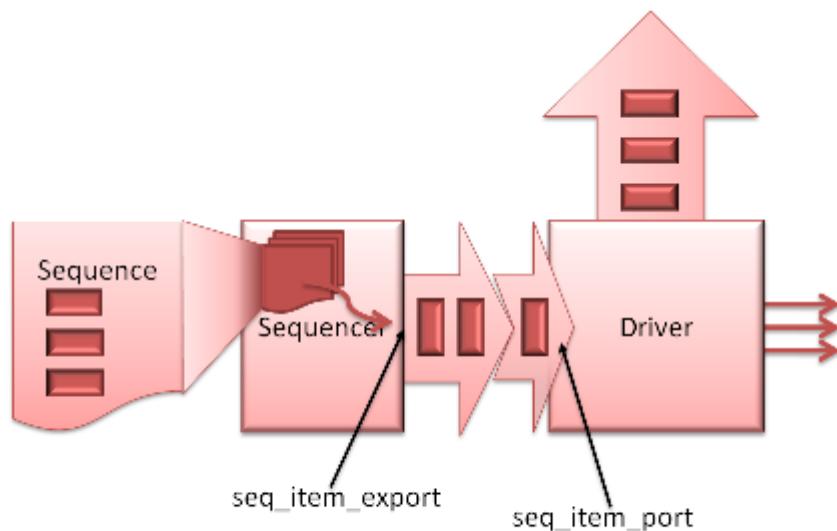
endclass : Driver

`endif

```

## Environment Updates

We will take the instance of Sequencer and Driver and connect them in Environment class.



© www.testbench.in

- 1) Declare handles to Driver and Sequencer.

```
Sequencer Seqncr;  
Driver Drvr;
```

- 2) In build method, construct Seqncr and Drvr object using create() method.

```
Drvr = Driver::type_id::create("Drvr",this);  
Seqncr = Sequencer::type_id::create("Seqncr",this);
```

- 2) In connect() method connect the sequencer seq\_item\_port to drivers seq\_item\_export.

```
Drvr.seq_item_port.connect(Seqncr.seq_item_export);
```

## Environment class code

```
`ifndef GUARD_ENV  
`define GUARD_ENV  
  
class Environment extends uvm_env;
```

```

`uvm_component_utils(Environment)

Sequencer Seqncr;
Driver Drvr;

function new(string name , uvm_component parent = null);
    super.new(name, parent);
endfunction: new

virtual function void build();
    super.build();
    uvm_report_info(get_full_name(),"START of build ",UVM_LOW);

    Drvr = Driver::type_id::create("Drvr",this);
    Seqncr = Sequencer::type_id::create("Seqncr",this);

    uvm_report_info(get_full_name(),"END of build ",UVM_LOW);
endfunction

virtual function void connect();
    super.connect();
    uvm_report_info(get_full_name(),"START of connect ",UVM_LOW);

    Drvr.seq_item_port.connect(Seqncr.seq_item_export);

    uvm_report_info(get_full_name(),"END of connect ",UVM_LOW);
endfunction

endclass : Environment
`endif

```

## Testcase Updates

We will update the testcase and run the simulation.

- 1) In the build() method, update the configuration address in the configuration object which in top module.

```

virtual function void build();
    super.build();

    cfg.device_add[0] = 0;
    cfg.device_add[1] = 1;
    cfg.device_add[2] = 2;
    cfg.device_add[3] = 3;

```

2) In the build() method itself, using set\_config\_object , configure the configuration object with the one which is in top module.

with this, the configuration object in Sequencer and Driver will be pointing to the one which in top module.

```
set_config_object("t_env.*","Configuration",cfg);
```

3) In the build method, using set\_config\_string, configure the default\_sequence of the sequencer to use the sequence which we defined.

```
set_config_string("*.Seqncr", "default_sequence", "Seq_device0_and_device1");
```

4) Set the sequencer count value to 2 .

```
set_config_int("*.Seqncr", "count",2);
```

5) Update the run() method to print the Sequencer details.

```
t_env.Seqncre.print();
```

### Testcase code

```

class test1 extends uvm_test;
`uvm_component_utils(test1)

Environment t_env;

function new (string name="test1", uvm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);

```

```

endfunction : new

virtual function void build();
    super.build();

    cfg.device_add[0] = 0;
    cfg.device_add[1] = 1;
    cfg.device_add[2] = 2;
    cfg.device_add[3] = 3;

    set_config_object("t_env.*","Configuration",cfg);
    set_config_string("*.Seqncr", "default_sequence", "Seq_device0_and_device1");
    set_config_int("*.Seqncr", "count",2);

endfunction

virtual task run ();
    t_env.Seqncr.print();

    #3000ns;
    global_stop_request();
endtask : run

endclass : test1

```

[Download the source code](#)

[uvm\\_switch\\_6.tar](#)

[Browse the code in uvm\\_switch\\_6.tar](#)

[Command to run the simulation](#)

VCS Users : make vcs

Questa Users: make questa

[Log report after simulation](#)

UVM\_INFO @ 0 [RNTST] Running test test1...  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] START of build  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] END of build  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] START of connect  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] END of connect

---

Name	Type	Size	Value
<hr/>			
Seqncr	Sequencer	-	Seqncr@14
--rsp_export	uvm_analysis_export	-	rsp_export@16
--seq_item_export	uvm_seq_item_pull_+	-	seq_item_export@40
--default_sequence	string	19	uvm_random_sequence
--count	integral	32	-1
--max_random_count	integral	32	'd10
--sequences	array	5	-
----[0]	string	19	uvm_random_sequence
----[1]	string	23	uvm_exhaustive_sequ+
----[2]	string	19	uvm_simple_sequence
----[3]	string	23	Seq_device0_and_dev+
----[4]	string	19	Seq_constant_length
--max_random_depth	integral	32	'd4
--num_last_reqs	integral	32	'd1
--num_last_rsps	integral	32	'd1
<hr/>			

UVM\_INFO @ 30: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     Start of reset\_dut() method  
 UVM\_INFO @ 70: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     End of reset\_dut() method  
 UVM\_INFO @ 70: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     Start of cfg\_dut() method  
 UVM\_INFO @ 110: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     Port 0 Address 00  
 UVM\_INFO @ 130: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     Port 1 Address 01  
 UVM\_INFO @ 150: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     Port 2 Address 02  
 UVM\_INFO @ 170: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     Port 3 Address 03  
 UVM\_INFO @ 190: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
     End of cfg\_dut() method

```

UVM_INFO @ 210: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...
UVM_INFO @ 590: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...
UVM_INFO @ 970: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...

```

--- UVM Report Summary ---

\*\* Report counts by severity

```

UVM_INFO : 16
UVM_WARNING : 0
UVM_ERROR : 0
UVM_FATAL : 0

```

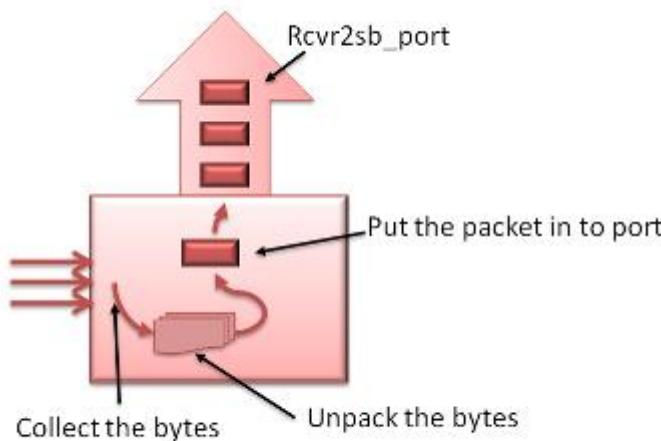
### **PHASE 7 RECEIVER**

In this phase, we will write a receiver and use the receiver in environment class to collect the packets coming from the switch output\_interface.

#### **Receiver**

Receiver collects the data bytes from the interface signal. And then unpacks the bytes in to packet using unpack\_bytes method and pushes it into Rcvr2Sb\_port for score boarding. Receiver class is written in Receiver.sv file.

Receiver class is defined by extending uvm\_component class. It will drive the received transaction to scoreboard using uvm\_analysis\_port.



- 1) Define Receiver class by extending uvm\_component.

```
`ifndef GUARD_RECEIVER
`define GUARD_RECEIVER

class Receiver extends uvm_component;

endclass : Receiver

`endif
```

2) Declare configuration class object.

```
Configuration cfg;
```

3) Declare an integer to hold the receiver number.

```
integer id;
```

4) Declare a virtual interface of dut out put side.

```
virtual output_interface.OP output_intf;
```

5) Declare analysis port which is used by receiver to send the received transaction to scoreboard.

```
uvm_analysis_port #(Packet) Rcvr2Sb_port;
```

6) Declare the utility macro. This utility macro provides the implementation of creat() and get\_type\_name() methods.

```
`uvm_component_utils(Receiver)
```

7) Define the constructor.

```
function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction : new
```

8) Define the build method and construct the Rcvr2Sb\_port.

```
virtual function void build();
```

```

super.build();
Rcvr2Sb_port = new("Rcvr2Sb", this);
endfunction : build

```

9) In the end\_of\_elaboration() method, get the configuration object using get\_config\_object and update the virtual interfaces.

```

virtual function void end_of_elaboration();
    uvm_object tmp;
    super.end_of_elaboration();
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
    output_intf = cfg.output_intf[id];
endfunction : end_of_elaboration

```

10) Define the run() method. This method collects the packets from the DUT output interface and unpacks it into high level transaction using transactions unpack\_bytes() method.

```

virtual task run();
    Packet pkt;
    fork
        forever
            begin
                // declare the queue and dynamic array here
                // so they are automatically allocated for every packet
                bit [7:0] bq[$],bytes[];
                repeat(2) @(posedge output_intf.clock);
                wait(output_intf.cb.ready)
                output_intf.cb.read <= 1;

                repeat(2) @(posedge output_intf.clock);
                while (output_intf.cb.ready)
                begin
                    bq.push_back(output_intf.cb.data_out);
                    @(posedge output_intf.clock);
                end
                bytes = new[bq.size()] (bq); // Copy queue into dyn array
                output_intf.cb.read <= 0;

```

```

@(posedge output_intf.clock);
uvm_report_info(get_full_name(),"Received packet ...",UVM_LOW);
pkt = new();
void'(pkt.unpack_bytes(bytes));
Rcvr2Sb_port.write(pkt);
end
join

endtask : run

```

### Receiver class source code

```

`ifndef GUARD_RECEIVER
`define GUARD_RECEIVER

class Receiver extends uvm_component;

virtual output_interface.OP output_intf;

Configuration cfg;

integer id;

uvm_analysis_port #(Packet) Rcvr2Sb_port;

`uvm_component_utils(Receiver)

function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction : new

virtual function void build();
    super.build();
    Rcvr2Sb_port = new("Rcvr2Sb", this);
endfunction : build

virtual function void end_of_elaboration();
    uvm_object tmp;

```

```

super.end_of_elaboration();
assert(get_config_object("Configuration",tmp));
$cast(cfg,tmp);
output_intf = cfg.output_intf[id];
endfunction : end_of_elaboration

virtual task run();
Packet pkt;
fork
forever
begin
    // declare the queue and dynamic array here
    // so they are automatically allocated for every packet
    bit [7:0] bq[$],bytes[];
    repeat(2) @(posedge output_intf.clock);
    wait(output_intf.cb.ready)
    output_intf.cb.read <= 1;

    repeat(2) @(posedge output_intf.clock);
    while (output_intf.cb.ready)
    begin
        bq.push_back(output_intf.cb.data_out);
        @(posedge output_intf.clock);
    end
    bytes = new[bq.size()] (bq); // Copy queue into dyn array

    output_intf.cb.read <= 0;
    @(posedge output_intf.clock);
    uvm_report_info(get_full_name(),"Received packet ...",UVM_LOW);
    pkt = new();
    void'(pkt.unpack_bytes(bytes));
    Rcvr2Sb_port.write(pkt);
end
join

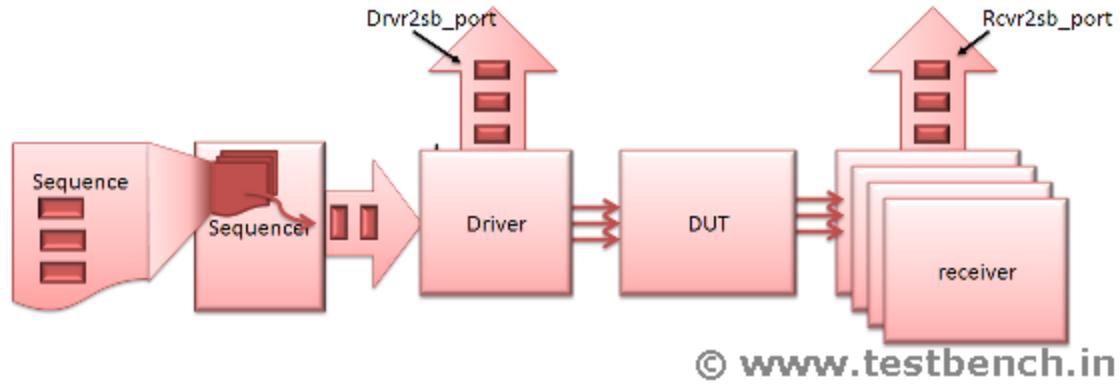
endtask : run

endclass : Receiver

```

## Environment Class Updates

We will update the Environment class and take instance of receiver and run the testcase.



- 1) Declare 4 receivers.

Receiver Rcvr[4];

- 2) In the build() method construct the Receivers using create() methods. Also update the id variable of the receiver object.

```
foreach(Rcvr[i]) begin
    Rcvr[i] = Receiver::type_id::create($psprintf("Rcvr%0d",i),this);
    Rcvr[i].id = i;
end
```

## Environment class source code

```
`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends uvm_env;

    `uvm_component_utils(Environment)

    Sequencer Seqncr;
    Driver Drvr;

    Receiver Rcvr[4];
```

```

function new(string name , uvm_component parent = null);
    super.new(name, parent);
endfunction: new

virtual function void build();
    super.build();
    uvm_report_info(get_full_name(),"START of build ",UVM_LOW);
    Drvr  = Driver::type_id::create("Drvr",this);
    Seqncr = Sequencer::type_id::create("Seqncr",this);

foreach(Rcvr[i]) begin
    Rcvr[i]  = Receiver::type_id::create($psprintf("Rcvr%0d",i),this);
    Rcvr[i].id = i;
end

    uvm_report_info(get_full_name(),"END of build ",UVM_LOW);
endfunction

virtual function void connect();
    super.connect();
    uvm_report_info(get_full_name(),"START of connect ",UVM_LOW);
    Drvr.seq_item_port.connect(Seqncr.seq_item_export);
    uvm_report_info(get_full_name(),"END of connect ",UVM_LOW);
endfunction

endclass : Environment
`endif

```

### [Download the Source Code](#)

[uvm\\_switch\\_7.tar](#)

[Browse the code in uvm\\_switch\\_7.tar](#)

### [Command to run the simulation](#)

VCS Users : make vcs

Questa Users: make questa

### [Log report after simulation](#)

UVM\_INFO @ 0 [RNTST] Running test test1...  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] START of build  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] END of build  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] START of connect  
 UVM\_INFO @ 0: uvm\_test\_top.t\_env [uvm\_test\_top.t\_env] END of connect

---

Name	Type	Size	Value
<hr/>			
Seqncr	Sequencer	-	Seqncr@14
--rsp_export	uvm_analysis_export	-	rsp_export@16
--seq_item_export	uvm_seq_item_pull_+	-	seq_item_export@40
--default_sequence	string	19	uvm_random_sequence
--count	integral	32	-1
--max_random_count	integral	32	'd10
--sequences	array	5	-
---[0]	string	19	uvm_random_sequence
---[1]	string	23	uvm_exhaustive_sequ+
---[2]	string	19	uvm_simple_sequence
---[3]	string	23	Seq_device0_and_dev+
---[4]	string	19	Seq_constant_length
--max_random_depth	integral	32	'd4
--num_last_reqs	integral	32	'd1
--num_last_rsp	integral	32	'd1
<hr/>			

UVM\_INFO @ 30: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
 Start of reset\_dut() method  
 UVM\_INFO @ 70: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
 End of reset\_dut() method  
 UVM\_INFO @ 70: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
 Start of cfg\_dut() method  
 UVM\_INFO @ 110: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
 Port 0 Address 00  
 UVM\_INFO @ 130: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
 Port 1 Address 01  
 UVM\_INFO @ 150: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
 Port 2 Address 02  
 UVM\_INFO @ 170: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]  
 Port 3 Address 03  
 UVM\_INFO @ 190: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]

```
    End of cfg_dut() method
UVM_INFO @ 210: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...
UVM_INFO @ 590: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...
UVM_INFO @ 610: uvm_test_top.t_env.Rcvr0 [uvm_test_top.t_env.Rcvr0]
    Received packet ...
UVM_INFO @ 970: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...
UVM_INFO @ 990: uvm_test_top.t_env.Rcvr0 [uvm_test_top.t_env.Rcvr0]
    Received packet ...
```

--- UVM Report Summary ---

\*\* Report counts by severity

```
UVM_INFO : 18
UVM_WARNING : 0
UVM_ERROR : 0
UVM_FATAL : 0
```

## **PHASE 8 SCOREBOARD**

In this phase we will see the scoreboard implementation.

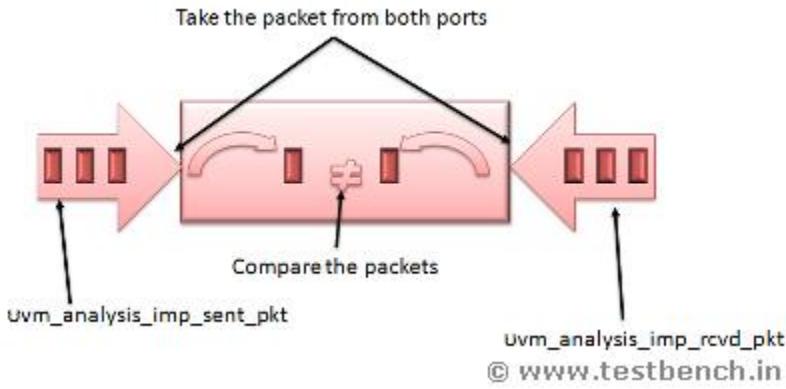
### **Scoreboard**

Scoreboard is implemented by extending uvm\_scorboard. For our requirement, we can use uvm\_in\_order\_comparator, but we will see develop our own scoreboard by extending uvm\_scorboard. Scoreboard has 2 analysis imports. One is used to for getting the packets from the driver and other from the receiver. Then the packets are compared and if they don't match, then error is asserted. For comparison, compare () method of the Packet class is used.

Implement the scoreboard in file Scoreboard.sv.

Steps to create a scoreboard:

- 1) Useing macro `uvm\_analysis\_imp\_decl(<\_portname>), to declare uvm\_analysis\_imp <\_portname> class.
- 2) The above macro, creates write\_<\_portname>(). This method has to be define as per our requirements.



1) We need 2 import, one for expected packet which is sent by driver and received packet which is coming from receiver.

Declare 2 imports using `uvm\_analysis\_imp\_decl macros should not be defined inside the class.

```
`uvm_analysis_imp_decl(_rcvd_pkt)
`uvm_analysis_imp_decl(_sent_pkt)
```

2) Declare a scoreboard by extending uvm\_scoreboard class.

```
class Scoreboard extends uvm_scoreboard;
endclass : Scoreboard
```

3) Declare the utility macro.

```
`uvm_component_utils(Scoreboard)
```

4) Declare a queue which stores the expected packets.

```
Packet exp_QUE[$];
```

5) Declare imports for getting expected packets and received packets.

```
uvm_analysis_imp_rcvd_pkt #(Packet,Scoreboard) Rcvr2Sb_port;
uvm_analysis_imp_sent_pkt #(Packet,Scoreboard) Drvr2Sb_port;
```

6) In the constructor, create objects for the above two declared imports.

```
function new(string name, uvm_component parent);
    super.new(name, parent);
```

```

Rcvr2Sb_port = new("Rcvr2Sb", this);
Drvrv2Sb_port = new("Drvrv2Sb", this);
endfunction : new

```

7) Define write\_sent\_pkt() method which was created by macro  
`uvm\_analysis\_imp\_decl(\_sent\_pkt).

In this method, store the received packet in the expected queue.

```

virtual function void write_sent_pkt(input Packet pkt);
    exp_que.push_back(pkt);
endfunction : write_sent_pkt

```

8) Define write\_rcvd\_pkt() method which was created by macro  
`uvm\_analysis\_imp\_decl(\_rcvd\_pkt)

In this method, get the transaction from the expected queue and compare.

```

virtual function void write_rcvd_pkt(input Packet pkt);
    Packet exp_pkt;
    pkt.print();
    if(exp_que.size())
        begin
            exp_pkt = exp_que.pop_front();
            exp_pkt.print();
            if(pkt.compare(exp_pkt))
                uvm_report_info(get_type_name(),
                    $psprintf("Sent packet and received packet matched"), UVM_LOW);
            else
                uvm_report_error(get_type_name(),
                    $psprintf("Sent packet and received packet mismatched"), UVM_LOW);
        end
        else
            uvm_report_error(get_type_name(),
                $psprintf("No more packets in the expected queue to compare"), UVM_LOW);
endfunction : write_rcvd_pkt

```

9) Define the report() method to print the Scoreboard information.

```

virtual function void report();
    uvm_report_info(get_type_name(),
        $psprintf("Scoreboard Report %s", this.sprint()), UVM_LOW);

```

```
endfunction : report
```

### Complete Scoreboard Code

```
`ifndef GUARD_SCOREBOARD
`define GUARD_SCOREBOARD

`uvm_analysis_imp_decl(_rcvd_pkt)
`uvm_analysis_imp_decl(_sent_pkt)

class Scoreboard extends uvm_scoreboard;
  `uvm_component_utils(Scoreboard)

  Packet exp_que[$];

  uvm_analysis_imp_rcvd_pkt #(Packet,Scoreboard) Rcvr2Sb_port;
  uvm_analysis_imp_sent_pkt #(Packet,Scoreboard) Drvr2Sb_port;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    Rcvr2Sb_port = new("Rcvr2Sb", this);
    Drvr2Sb_port = new("Drvr2Sb", this);
  endfunction : new

  virtual function void write_rcvd_pkt(input Packet pkt);
    Packet exp_pkt;
    pkt.print();

    if(exp_que.size())
    begin
      exp_pkt = exp_que.pop_front();
      exp_pkt.print();
      if (pkt.compare(exp_pkt))
        uvm_report_info(get_type_name(),
                      $psprintf("Sent packet and received packet matched"), UVM_LOW);
      else
        uvm_report_error(get_type_name(),
                         $psprintf("Sent packet and received packet mismatched"), UVM_LOW);
    end
    else
      uvm_report_error(get_type_name(),
```

```

    $psprintf("No more packets to in the expected queue to compare"), UVM_LOW);
endfunction : write_rcvd_pkt

virtual function void write_sent_pkt(input Packet pkt);
    exp_que.push_back(pkt);
endfunction : write_sent_pkt

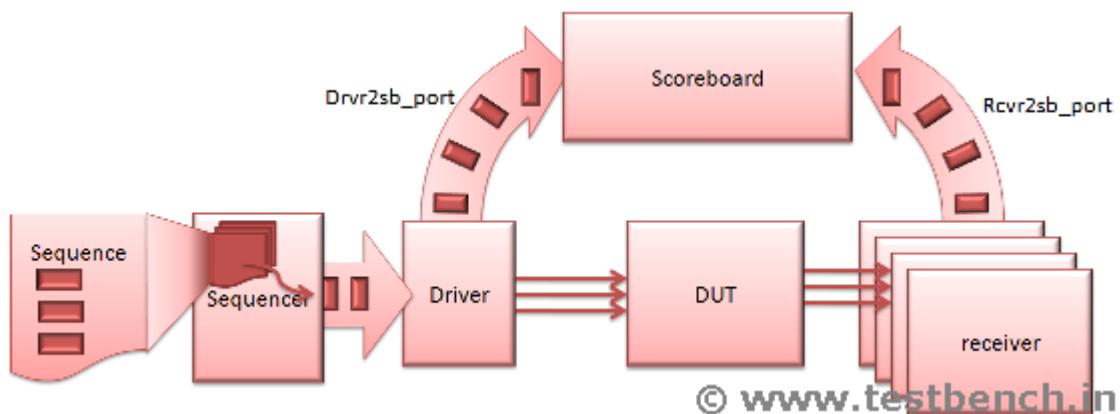
virtual function void report();
    uvm_report_info(get_type_name(),
        $psprintf("Scoreboard Report %s", this.sprint()), UVM_LOW);
endfunction : report

endclass : Scoreboard
`endif

```

### Environment Class Updates

We will take the instance of scoreboard in the environment and connect its ports to driver and receiver ports.



- 1) Declare scoreboard object.

Scoreboard Sbd;

- 2) Construct the scoreboard object using create() method in build() method.

```
Sbd = Scoreboard::type_id::create("Sbd",this);
```

- 3) In connect() method, connect the driver and receiver ports to scoreboard.

```

Drvrv.Drvr2Sb_port.connect(Sbd.Drvr2Sb_port);

foreach(Rcvr[i])
    Rcvr[i].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);

```

### Environemnt class code

```

`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends uvm_env;

    `uvm_component_utils(Environment)

    Sequencer Seqnrc;
    Driver Drvr;
    Receiver Rcvr[4];

    Scoreboard Sbd;

    function new(string name , uvm_component parent = null);
        super.new(name, parent);
    endfunction: new

    virtual function void build();
        super.build();
        uvm_report_info(get_full_name(),"START of build ",UVM_LOW);

        Drvr = Driver::type_id::create("Drvr",this);
        Seqnrc = Sequencer::type_id::create("Seqnrc",this);

        foreach(Rcvr[i]) begin
            Rcvr[i] = Receiver::type_id::create($psprintf("Rcvr%0d",i),this);
            Rcvr[i].id = i;
        end

        Sbd = Scoreboard::type_id::create("Sbd",this);

        uvm_report_info(get_full_name(),"END of build ",UVM_LOW);
    end

```

```

endfunction

virtual function void connect();
    super.connect();
    uvm_report_info(get_full_name(),"START of connect ",UVM_LOW);

    Drvr.seq_item_port.connect(Seqncr.seq_item_export);

    Drvr.Drvr2Sb_port.connect(Sbd.Drvr2Sb_port);

foreach(Rcvr[i])
    Rcvr[i].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);

    uvm_report_info(get_full_name(),"END of connect ",UVM_LOW);
endfunction

endclass : Environment
`endif

```

### Download the Source Code

[uvm\\_switch\\_8.tar](#)

[Browse the code in uvm\\_switch\\_8.tar](#)

### Command to run the simulation

VCS Users : make vcs

Questa Users: make questa

### Log report after simulation

```

UVM_INFO @ 0 [RNTST] Running test test1...
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] START of build
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] END of build
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] START of connect
UVM_INFO @ 0: uvm_test_top.t_env [uvm_test_top.t_env] END of connect
-----
```

Name	Type	Size	Value
------	------	------	-------

```
-----
Seqncr      Sequencer      -      Seqncr@14
rsp_export   uvm_analysis_export -      rsp_export@16
seq_item_export  uvm_seq_item_pull_+ -  seq_item_export@40
default_sequence string      19  uvm_random_sequence
count        integral      32      -1
max_random_count integral      32      'd10
sequences     array       5       -
[0]          string      19  uvm_random_sequence
[1]          string      23  uvm_exhaustive_sequ+
[2]          string      19  uvm_simple_sequence
[3]          string      23  Seq_device0_and_dev+
[4]          string      19  Seq_constant_length
max_random_depth integral      32      'd4
num_last_reqs    integral      32      'd1
num_last_rsps    integral      32      'd1
-----
```

```
UVM_INFO @ 30: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Start of reset_dut() method
UVM_INFO @ 70: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    End of reset_dut() method
UVM_INFO @ 70: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Start of cfg_dut() method
UVM_INFO @ 110: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Port 0 Address 00
UVM_INFO @ 130: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Port 1 Address 01
UVM_INFO @ 150: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Port 2 Address 02
UVM_INFO @ 170: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Port 3 Address 03
UVM_INFO @ 190: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    End of cfg_dut() method
UVM_INFO @ 210: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...
UVM_INFO @ 590: uvm_test_top.t_env.Drvr [uvm_test_top.t_env.Drvr]
    Driving packet ...
UVM_INFO @ 610: uvm_test_top.t_env.Rcvr0 [uvm_test_top.t_env.Rcvr0]
    Received packet ...
UVM_INFO @ 610: uvm_test_top.t_env.Sbd [Scoreboard]
```

Sent packet and received packet matched

UVM\_INFO @ 970: uvm\_test\_top.t\_env.Drvr [uvm\_test\_top.t\_env.Drvr]

Driving packet ...

UVM\_INFO @ 990: uvm\_test\_top.t\_env.Rcvr0 [uvm\_test\_top.t\_env.Rcvr0]

Received packet ...

UVM\_INFO @ 990: uvm\_test\_top.t\_env.Sbd [Scoreboard]

Sent packet and received packet matched

UVM\_INFO @ 1000: uvm\_test\_top.t\_env.Sbd [Scoreboard]

Scoreboard Report

---

Name	Type	Size	Value
Sbd	Scoreboard	-	Sbd@52
Drvrv2Sb	uvm_analysis_imp_s+	-	Drvrv2Sb@56
Rcvrv2Sb	uvm_analysis_imp_r+	-	Rcvrv2Sb@54

---

--- UVM Report Summary ---

\*\* Report counts by severity

UVM\_INFO : 21

UVM\_WARNING : 0

UVM\_ERROR : 0

UVM\_FATAL : 0

## **INDEX**

### **INTRODUCTION**

#### **SPECIFICATION**

Switch Specification

Packet Format

Configuration

Interface Specification

#### **VERIFICATION PLAN**

Overview

Feature Extraction

Stimulus Generation Plan

Verification Environment

#### **PHASE 1 TOP**

Interface

Top Module

#### **PHASE 2 CONFIGURATION**

Configuration

Updates To Top Module

#### **PHASE 3 ENVIRONMENT N TESTCASE**

Environment

Testcase

#### **PHASE 4 PACKET**

Packet

Test The Transaction Implementation

#### **PHASE 5 SEQUENCER N SEQUENCE**

Sequencer

Sequence

#### **PHASE 6 DRIVER**

Driver

Environment Updates

Testcase Updates

#### **PHASE 7 RECEIVER**

Receiver

Environment Class Updates

#### **PHASE 8 SCOREBOARD**

Scoreboard

Environment Class Updates

## **INTRODUCTION**

In this tutorial, we will verify the Switch RTL core using OVM in SystemVerilog. Following are the steps we follow to verify the Switch RTL core.

- 1) Understand the specification
- 2) Developing Verification Plan
- 3) Building the Verification Environment. We will build the Environment in Multiple phases, so it will be easy for you to learn step by step.

In this verification environment, I will not use agents and monitors to make this tutorial simple and easy.

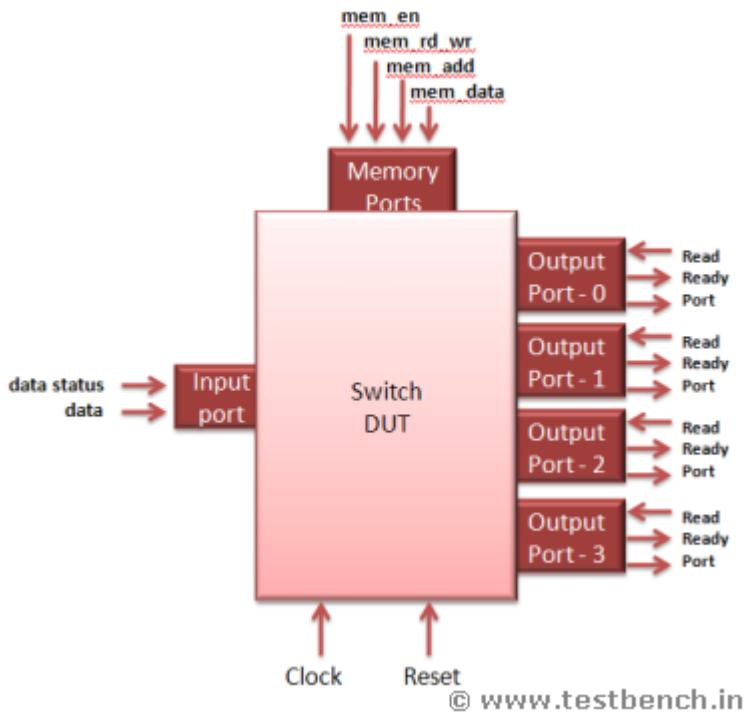
- Phase 1) We will develop the interfaces, and connect it to DUT in top module.
- Phase 2) We will develop the Configuration class.
- Phase 3) We will develop the Environment class and Simple testcase and simulate them.
- Phase 4) We will develop packet class based on the stimulus plan. We will also write a small code to test the packet class implementation.
- Phase 5) We will develop sequencer and a sample sequences.
- Phase 6) We will develop driver and connect it to the Sequencer in to environment.
- Phase 7) We will develop receiver and instantiate in environment.
- Phase 8) We will develop scoreboard which does the comparison of the expected packet with the actual packet received from the DUT and connect it to driver and receiver in Environment class.

I would like to thank to Vishnu Prashant(Vitesse) for teaching me OVM.

## **SPECIFICATION**

### **Switch Specification:**

This is a simple switch. Switch is a packet based protocol. Switch drives the incoming packet which comes from the input port to output ports based on the address contained in the packet. The switch has a one input port from which the packet enters. It has four output ports where the packet is driven out.



### Packet Format:

Packet contains 3 parts. They are Header, data and frame check sequence.

Packet width is 8 bits and the length of the packet can be between 4 bytes to 259 bytes.

#### Packet header:

Packet header contains three fields DA, SA and length.

② DA: Destination address of the packet is of 8 bits. The switch drives the packet to respective ports based on this destination address of the packets. Each output port has 8-bit unique port address. If the destination address of the packet matches the port address, then switch drives the packet to the output port.

② SA: Source address of the packet from where it originate. It is 8 bits.

② Length: Length of the data is of 8 bits and from 0 to 255. Length is measured in terms of bytes.

If Length = 0, it means data length is 0 bytes

If Length = 1, it means data length is 1 bytes

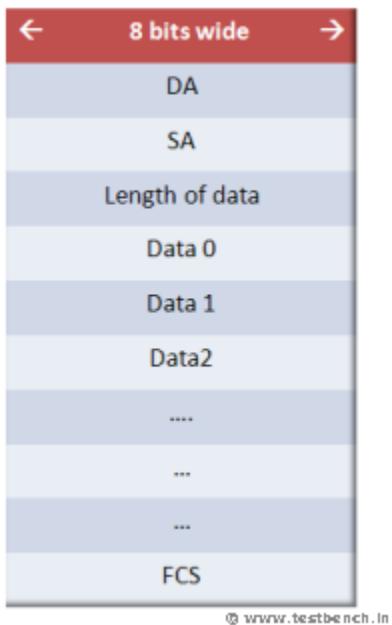
If Length = 2, it means data length is 2 bytes

If Length = 255, it means data length is 255 bytes

② Data: Data should be in terms of bytes and can take anything.

## FCS: Frame check sequence

This field contains the security check of the packet. It is calculated over the header and data.



### Configuration:

Switch has four output ports. These output ports address have to be configured to a unique address. Switch matches the DA field of the packet with this configured port address and sends the packet on to that port. Switch contains a memory. This memory has 4 locations, each can store 8 bits. To configure the switch port address, memory write operation has to be done using memory interface. Memory address (0,1,2,3) contains the address of port(0,1,2,3) respectively.

### Interface Specification:

The Switch has one input Interface, from where the packet enters and 4 output interfaces from where the packet comes out and one memory interface, through the port address can be configured. Switch also has a clock and asynchronous reset signal.

### MEMORY INTERFACE:

Through memory interfaced output port address are configured. It accepts 8 bit data to be written to memory. It has 8 bit address inputs. Address 0,1,2,3 contains the address of the port 0,1,2,3 respectively.

There are 4 input signals to memory interface. They are

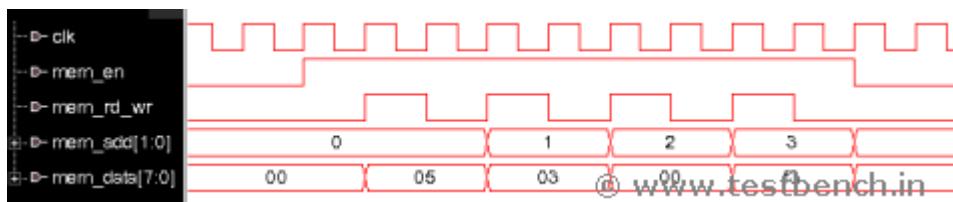
**input mem\_en;**  
**input mem\_rd\_wr;**

```
input [1:0] mem_add;  
input [7:0] mem_data;
```

All the signals are active high and are synchronous to the positive edge of clock signal.

To configure a port address,

1. Assert the mem\_en signal.
2. Assert the mem\_rd\_wr signal.
3. Drive the port number (0 or 1 or 2 or 3) on the mem\_add signal
4. Drive the 8 bit port address on to mem\_data signal.



## INPUT PORT

Packets are sent into the switch using input port.

All the signals are active high and are synchronous to the positive edge of clock signal.

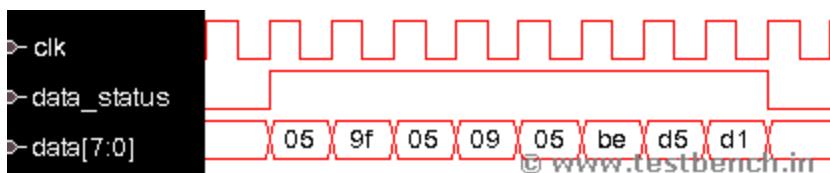
**input** port has **2 input** signals. They are

```
input [7:0] data;
```

```
input data_status;
```

To send the packet in to switch,

1. Assert the data\_status signal.
2. Send the packet on the data signal byte by byte.
3. After sending all the data bytes, deassert the data\_status signal.
4. There should be at least 3 clock cycles difference between packets.



## OUTPUT PORT

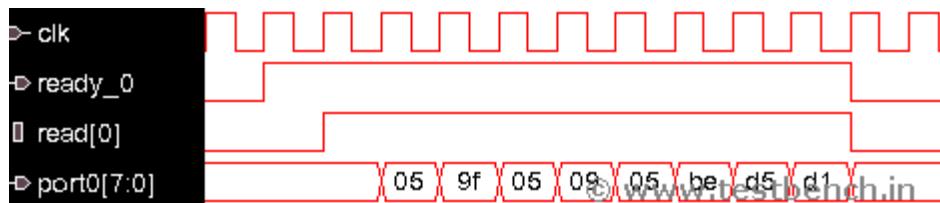
Switch sends the packets out using the output ports. There are 4 ports, each having data, ready and read signals. All the signals are active high and are synchronous to the positive edge of clock signal.

Signal list is

```
output [7:0] port0;
output [7:0] port1;
output [7:0] port2;
output [7:0] port3;
output    ready_0;
output    ready_1;
output    ready_2;
output    ready_3;
input     read_0;
input     read_1;
input     read_2;
input     read_3;
```

When the data is ready to be sent out from the port, switch asserts ready\_\* signal high indicating that data is ready to be sent.

If the read\_\* signal is asserted, when ready\_\* is high, then the data comes out of the port\_\* signal after one clock cycle.



## RTL code:

RTL code is attached with the tar files. From the Phase 1, you can download the tar files.

## VERIFICATION PLAN

### Overview

This Document describes the Verification Plan for Switch. The Verification Plan is based on System Verilog Hardware Verification Language. The methodology used for Verification is Constraint random coverage driven verification.

## **Feature Extraction**

This section contains list of all the features to be verified.

1)

ID: Configuration

Description: Configure all the 4 port address with unique values.

2)

ID: Packet DA

Description: DA field of packet should be any of the port address. All the 4 port address should be used.

3)

ID : Packet payload

Description: Length can be from 1 to 255. Send packets with all the lengths.

4)

ID: Length

Description:

Length field contains length of the payload.

5)

ID: FCS

Description:

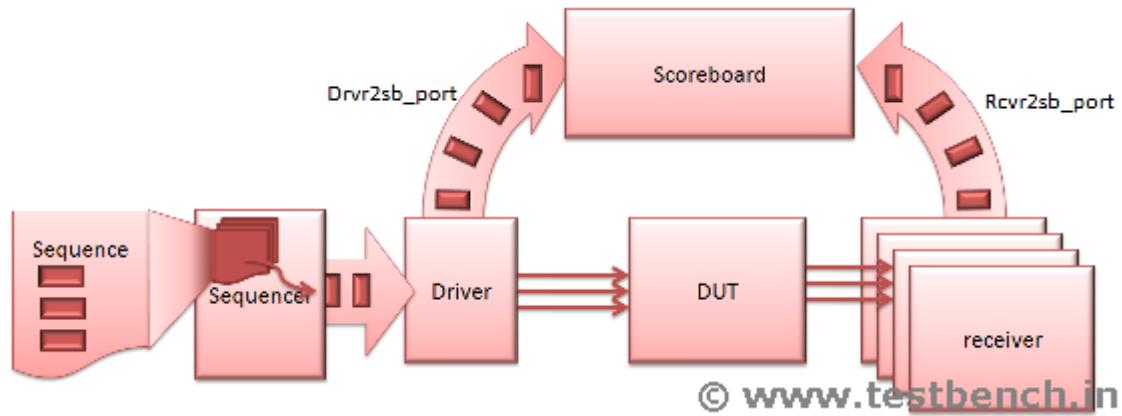
Good FCS: Send packet with good FCS.

Bad FCS: Send packet with corrupted FCS.

## **Stimulus Generation Plan**

- 1) Packet DA: Generate packet DA with the configured address.
- 2) Payload length: generate payload length ranging from 2 to 255.
- 3) Generate good and bad FCS.

## Verification Environment



### PHASE 1 TOP

In phase 1,

- 1) We will write SystemVerilog Interfaces for input port, output port and memory port.
- 2) We will write Top module where testcase and DUT instances are done.
- 3) DUT and interfaces are connected in top module.
- 4) We will implement Clock generator in top module.

### Interface

In the interface.sv file, declare the 3 interfaces in the following way.

All the interfaces has clock as input.

All the signals in interface are wire type.

All the signals are synchronized to clock except reset in clocking block.

This approach will avoid race conditions between the design and the verification environment.

Define the set-up and hold time using parameters.

Signal directional w.r.t TestBench is specified with modport.

### Interface Source Code

```
`ifndef GUARD_INTERFACE  
`define GUARD_INTERFACE  
///////////////////////////////  
// Interface declaration for the memory//  
///////////////////////////////
```

```
interface mem_interface(input bit clock);  
    parameter setup_time = 5ns;  
    parameter hold_time = 3ns;  
    wire [7:0] mem_data;  
    wire [1:0] mem_add;  
    wire mem_en;
```

```

wire      mem_rd_wr;
clocking cb@(posedge clock);
  default input #setup_time output #hold_time;
  output    mem_data;
  output    mem_add;
  output    mem_en;
  output    mem_rd_wr;
endclocking:cb

modport MEM(clocking cb,input clock);
endinterface :mem_interface

///////////////////////////////
// Interface for the input side of switch.//
// Reset signal is also passed here.   //
///////////////////////////////

interface input_interface(input bit clock);
  parameter setup_time = 5ns;
  parameter hold_time = 3ns;
  wire      data_status;
  wire      [7:0] data_in;
  reg       reset;

  clocking cb@(posedge clock);
    default input #setup_time output #hold_time;
    output    data_status;
    output    data_in;
endclocking:cb
  modport IP(clocking cb,output reset,input clock);
endinterface:input_interface

///////////////////////////////
// Interface for the output side of the switch.//
// output_interface is for only one output port//
///////////////////////////////

interface output_interface(input bit clock);
  parameter setup_time = 5ns;
  parameter hold_time = 3ns;
  wire      [7:0] data_out;
  wire      ready;
  wire      read;

```

```

clocking cb@(posedge clock);
  default input #setup_time output #hold_time;
  input  data_out;
  input  ready;
  output read;
endclocking:cb
  modport OP(clocking cb,input clock);
endinterface:output_interface
///////////
`endif

```

## Top Module

The modules that are included in the source text but are not instantiated are called top modules. This module is the highest scope of modules. Generally this module is named as "top" and referenced as "top module". Module name can be anything.

This top-level module will contain the design portion of the simulation.

Do the following in the top module:

1) The first step is to import the ovm packages

```
`include "ovm.svh"
import ovm_pkg::*;
```

2) Generate the clock signal.

```
bit Clock;
initial
begin
#20;
forever #10 Clock = ~Clock;
end
```

2) Do the instances of memory interface.

```
mem_interface mem_intf(Clock);
```

3) Do the instances of input interface.

```
input_interface input_intf(Clock);
```

4) There are 4 output ports. So do 4 instances of output\_interface.

```
output_interface output_intf[4](Clock);
```

5) Connect all the interfaces and DUT. The design which we have taken is in verilog. So Verilog DUT instance is connected signal by signal.

```
switch DUT (.clk(Clock),
.reset(input_intf.reset),
.data_status(input_intf.data_status),
.data(input_intf.data_in),
.port0(output_intf[0].data_out),
```

```

    .port1(output_intf[1].data_out),
    .port2(output_intf[2].data_out),
    .port3(output_intf[3].data_out),
    .ready_0(output_intf[0].ready),
    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),
    .mem_en(mem_intf.mem_en),
    .mem_rd_wr(mem_intf.mem_rd_wr),
    .mem_add(mem_intf.mem_add),
    .mem_data(mem_intf.mem_data));

```

### Top module Scource Code

```

`ifndef GUARD_TOP
#define GUARD_TOP
///////////////////////////////
// Importing OVM Packages          //
///////////////////////////////

`include "ovm.svh"
import ovm_pkg::*;

typedef class Configuration;

module top();

///////////////////////////////
// Clock Declaration and Generation      //
///////////////////////////////
bit Clock;

initial
begin
#20;
forever #10 Clock = ~Clock;

```

```

end
///////////
// Memory interface instance          //
///////////

mem_interface mem_intf(Clock);

///////////
// Input interface instance          //
///////////

input_interface input_intf(Clock);

///////////
// output interface instance         //
///////////

output_interface output_intf[4](Clock);

///////////
// DUT instance and signal connection //
///////////

switch DUT (.clk(Clock),
    .reset(input_intf.reset),
    .data_status(input_intf.data_status),
    .data(input_intf.data_in),
    .port0(output_intf[0].data_out),
    .port1(output_intf[1].data_out),
    .port2(output_intf[2].data_out),
    .port3(output_intf[3].data_out),
    .ready_0(output_intf[0].ready),
    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),
    .mem_en(mem_intf.mem_en),

```

```

.mem_rd_wr(mem_intf.mem_rd_wr),
.mem_add(mem_intf.mem_add),
.mem_data(mem_intf.mem_data));

```

**endmodule : top**

`endif

[Download the files:](#)

[ovm\\_switch\\_1.tar](#)

[Browse the code in ovm\\_switch\\_1.tar](#)

[Command to compile](#)

```
vcs -sverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv
qverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv
```

## **PHASE 2 CONFIGURATION**

In this phase we will implement the configuration class. All the requirements of the testbench configurations will be declared inside this class. Virtual interfaces required by verification components driver and receiver for connecting to DUT are declared in this class. We will also declare 4 variables which will hold the port address of the DUT.

ovm\_object does not have the simulation phases and can be used in get\_config\_object and set\_config\_object method. So we will implement the configuration class by extending ovm\_object.

### **Configuration**

1) Define configuration class by extending ovm\_object

```
`ifndef GUARD_CONFIGURATION
`define GUARD_CONFIGURATION
class Configuration extends ovm_object;
endclass : Configuration
`endif
```

2) Declare All the interfaces which are required in this verification environment.

```
virtual input_interface.IP input_intf;
virtual mem_interface.MEM mem_intf;
virtual output_interface.OP output_intf[4];
```

3) Declare 4 variables which holds the device port address.

```

bit [7:0] device0_add ;
bit [7:0] device1_add ;
bit [7:0] device2_add ;
bit [7:0] device3_add ;

```

4) ovm\_object required to define the ovm\_object::creat() method.

ovm\_object::create method allocates a new object of the same type as this object and returns it via a base ovm\_object handle.

In create method, we have to construct a new object of configuration class and update all the important fields and return it.

```

virtual function ovm_object create(string name="");
    Configuration t = new();
    t.device0_add = this.device0_add;
    t.device1_add = this.device1_add;
    t.device2_add = this.device2_add;
    t.device3_add = this.device3_add;
    t.input_intf = this.input_intf;
    t.mem_intf = this.mem_intf;
    t.output_intf = this.output_intf;

    return t;
endfunction : create

```

#### Configuration class source code

```

`ifndef GUARD_CONFIGURATION
`define GUARD_CONFIGURATION
class Configuration extends ovm_object;
    virtual input_interface.IP input_intf;
    virtual mem_interface.MEM mem_intf;
    virtual output_interface.OP output_intf[4];
    bit [7:0] device0_add ;
    bit [7:0] device1_add ;
    bit [7:0] device2_add ;
    bit [7:0] device3_add ;

    virtual function ovm_object create(string name="");
        Configuration t = new();
        t.device0_add = this.device0_add;
        t.device1_add = this.device1_add;
        t.device2_add = this.device2_add;
        t.device3_add = this.device3_add;
        t.input_intf = this.input_intf;

```

```

t.mem_intf = this.mem_intf;
t.output_intf = this.output_intf;

return t;
endfunction : create

endclass : Configuration
`endif

```

### Updates To Top Module

In top module we will create an object of the above defined configuration class and update the interfaces so that all the verification components can access to physical interfaces in top module using configuration class object.

1) Declare a Configuration class object

```
Configuration cfg;
```

2) Construct the configuration object and update the interfaces.

```
initial begin
```

```

cfg = new();
cfg.input_intf = input_intf;
cfg.mem_intf = mem_intf;
cfg.output_intf = output_intf;

```

3) In top module , we have to call the run\_test() method.

```
run_test();
```

### Top module updates

```

typedef class Configuration;

module top();
///////////////////////////////
// Clock Declaration and Generation      //
///////////////////////////////
bit Clock;

initial
begin
#20;
forever #10 Clock = ~Clock;
end

```

```

///////////
// Memory interface instance          //
///////////
mem_interface mem_intf(Clock);
///////////
// Input interface instance          //
///////////
input_interface input_intf(Clock);
///////////
// output interface instance         //
///////////
output_interface output_intf[4](Clock);

```

```

///////////
// Creat Configuration and Strart the run_test//
/////////

```

Configuration cfg;

```

initial begin
  cfg = new();
  cfg.input_intf = input_intf;
  cfg.mem_intf = mem_intf;
  cfg.output_intf = output_intf;

  run_test();
end

```

```

///////////
// DUT instance and signal connection      //
/////////
switch DUT (.clk(Clock),
  .reset(input_intf.reset),
  .data_status(input_intf.data_status),
  .data(input_intf.data_in),
  .port0(output_intf[0].data_out),
  .port1(output_intf[1].data_out),
  .port2(output_intf[2].data_out),
  .port3(output_intf[3].data_out),
  .ready_0(output_intf[0].ready),

```

```

    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),
    .mem_en(mem_intf.mem_en),
    .mem_rd_wr(mem_intf.mem_rd_wr),
    .mem_add(mem_intf.mem_add),
    .mem_data(mem_intf.mem_data));
endmodule : top
`endif

```

### Download the source code

[ovm\\_switch\\_2.tar](#)

[Browse the code in ovm\\_switch\\_2.tar](#)

### Command to compile

```
vcs -sverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv
qverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv
```

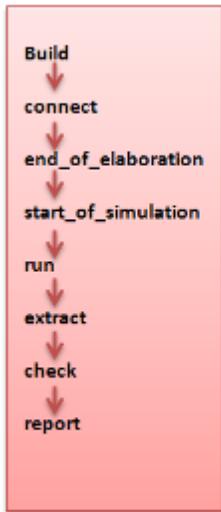
### PHASE 3 ENVIRONMENT N TESTCASE

In the phase we will implement the skeleton for environment class.

We will declare virtual interfaces and Extend Required Environment class virtual methods.

We will also implement a simple testcase and run the simulation.

Environment class is used to implement verification environments in OVM. It is extension on ovm\_env class. The testbench simulation needs some systematic flow like building the components, connection the components, starting the components etc. ovm\_env base class has methods formalize the simulation steps. All methods are declared as virtual methods.



© www.balbanch.in

We will not implement all the ovm\_env virtual methods in this phase but will we print messages from these methods which are required for this example to understand the simulation execution.

Testcase contains the instance of the environment class. This testcase Creates a Environment object and defines the required test specific functionality.

Verification environment contains the declarations of the virtual interfaces. These virtual interfaces are pointed to the physical interfaces which are declared in the top module. These virtual interfaces are made to point to physical interface in the testcase.

## Environment

1) Extend ovm\_env class to define Environment class.

```

`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends ovm_env;

endclass : Environment

`endif

```

2) Declare the utility macro. This utility macro provides the implementation of create() and get\_type\_name() methods.

```
`ovm_component_utils(Environment)
```

3) Define the constructor. In the constructor, call the super methods and pass the parent object. Parent is the object in which environment is instantiated.

```
function new(string name , ovm_component parent = null);  
    super.new(name, parent);  
endfunction: new
```

4) Define build method. In build method, just print messages and super.build() must be called. This method is automatically called. Build is the first phase in simulation. This phase is used to construct the child components of the current class.

```
function void build();  
    super.build();  
  
    ovm_report_info(get_full_name(),"START of build ",OVM_LOW);  
  
    ovm_report_info(get_full_name(),"END of build ",OVM_LOW);  
  
endfunction
```

5) Define connect method. In connect method, just print messages and super.connect() must be called. This method is called automatically after the build() method is called. This method is used for connecting port and exports.

```
function void connect();  
    super.connect();  
    ovm_report_info(get_full_name(),"START of connect ",OVM_LOW);  
  
    ovm_report_info(get_full_name(),"END of connect ",OVM_LOW);  
endfunction
```

### Environment class Source Code

```
`ifndef GUARD_ENV  
`define GUARD_ENV
```

```

class Environment extends ovm_env;

`ovm_component_utils(Environment)

function new(string name , ovm_component parent = null);
    super.new(name, parent);
endfunction: new

function void build();
    super.build();

    ovm_report_info(get_full_name(),"START of build ",OVM_LOW);

    ovm_report_info(get_full_name(),"END of build ",OVM_LOW);

endfunction

function void connect();
    super.connect();
    ovm_report_info(get_full_name(),"START of connect ",OVM_LOW);

    ovm_report_info(get_full_name(),"END of connect ",OVM_LOW);
endfunction

endclass : Environment
`endif

```

## Testcase

Now we will implement testcase. In OVM, testcases are implemented by extending ovm\_test. Using ovm\_test , provides the ability to select which test to execute using the OVM\_TESTNAME command line option or argument to the ovm\_root::run\_test task. We will use OVM\_TESTNAME command line argument.

- 1) Define a testcase by extending ovm\_test class.

```

class test1 extends ovm_test;

endclass

```

2) Declare the utility macro.

```
`ovm_component_utils(test1)
```

3) Take the instance of Environment.

```
Environment t_env ;
```

4) Define the constructor method.

In this method, construct the environment class object and dont forget to pass the parent argument.

```
function new (string name="test1", ovm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new
```

5) Define run() method.

run() method is the only task which is time consuming. After completing the start\_of\_simulation() phase , this method is called.

To terminate this task, we will use global\_stop\_request().

As we dont have anything now to write in this testcase, just call the global\_stop\_request() after some delay.

```
task run ();
#1000;
global_stop_request();
endtask : run
```

With this, for the first time, we can do the simulation.

#### Testcase Source code

```
class test1 extends ovm_test;
```

```
`ovm_component_utils(test1)
```

```
Environment t_env ;
```

```

function new (string name="test1", ovm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new

task run ();
#1000;
global_stop_request();
endtask : run

endclass : test1

```

### Download the Source Code

[ovm\\_switch\\_3.tar](#)

[Browse the code in ovm\\_switch\\_3.tar](#)

### Command to run the simulation

```

vcs -sverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1
qverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1

```

### Log report after simulation

```

OVM_INFO @ 0 [RNTST] Running test test1...
OVM_INFO @ 0: ovm_test_top.t_env [ovm_test_top.t_env] START of build
OVM_INFO @ 0: ovm_test_top.t_env [ovm_test_top.t_env] END of build
OVM_INFO @ 0: ovm_test_top.t_env [ovm_test_top.t_env] START of connect
OVM_INFO @ 0: ovm_test_top.t_env [ovm_test_top.t_env] END of connect

```

--- OVM Report Summary ---

\*\* Report counts by severity

OVM\_INFO : 5

OVM\_WARNING : 0

OVM\_ERROR : 0

```
OVM_FATAL : 0
** Report counts by id
[RNTST      ] 1
[ovm_test_top.t_env ] 4
```

## **PHASE 4 PACKET**

In this Phase, we will develop Transaction as per the verification plan. We will define required methods and constraints. We will also develop a small logic to test our implementation of this class.

### **Packet**

We will write the packet class in Packet.sv file. Packet class variables and constraints have been derived from stimulus generation plan.

One way to model Packet is by extending ovm\_sequence\_item. ovm\_sequence\_item provides basic functionality for sequence items and sequences to operate in a sequence mechanism. Packet class should be able to generate all possible packet types randomly. To define copy, compare, record, print and sprint methods, we will use OVM field macros. For packing and Unpacking, we will define the logic and not use the field macros.

#### Revisit Stimulus Generation Plan

- 1) Packet DA: Generate packet DA with the configured address.
- 2) Payload length: generate payload length ranging from 2 to 255.
- 3) Generate good and bad FCS.

- 1) Define enumerated type data for fcs.

```
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;
```

- 2) Define transaction by extending ovm\_sequence\_item.

```
class Packet extends ovm_sequence_item;
```

```
endclass : Packet
```

3) Define all the fields as rand variables.

```
rand fcs_kind_t fcs_kind;  
  
rand bit [7:0] length;  
rand bit [7:0] da;  
rand bit [7:0] sa;  
rand bit [7:0] data[];  
rand byte fcs;
```

4) Define constraints to constraint payload size of data.

```
constraint payload_size_c { data.size inside { [2 : 255]};}  
  
constraint length_c { length == data.size; }
```

5) Define the constructor method.

```
function new(string name = "");  
    super.new(name);  
endfunction : new
```

6) In post\_randomize() , define the fcs value based on fcs\_kind.

```
function void post_randomize();  
    if(fcs_kind == GOOD_FCS)  
        fcs = 8'b0;  
    else  
        fcs = 8'b1;  
    fcs = cal_fcs();  
endfunction : post_randomize
```

7) Define cal\_fcs() method which computes the fcs value.

```
virtual function byte cal_fcs;  
    integer i;
```

```

byte result ;
result = 0;
result = result ^ da;
result = result ^ sa;
result = result ^ length;
for (i = 0;i< data.size;i++)
result = result ^ data[i];
result = fcs ^ result;
return result;
endfunction : cal_fcs

```

8) Using ovm\_field\_\* macros, define transaction required method.

We will define packing and unpacking methods manually, so use OVM\_NOPACK for excluding atomic creation of packing and un packing method.

```

`ovm_object_utils_begin(Packet)
 `ovm_field_int(da, OVM_ALL_ON|OVM_NOPACK)
 `ovm_field_int(sa, OVM_ALL_ON|OVM_NOPACK)
 `ovm_field_int(length, OVM_ALL_ON|OVM_NOPACK)
 `ovm_field_array_int(data, OVM_ALL_ON|OVM_NOPACK)
 `ovm_field_int(fcs, OVM_ALL_ON|OVM_NOPACK)
`ovm_object_utils_end

```

9) Define do\_pack() method which does the packing operation.

```

function void do_pack(ovm_packer packer);
  super.do_pack(packer);
  packer.pack_field_int(da,$bits(da));
  packer.pack_field_int(sa,$bits(sa));
  packer.pack_field_int(length,$bits(length));
  foreach(data[i])
    packer.pack_field_int(data[i],8);
  packer.pack_field_int(fcs,$bits(fcs));
endfunction : do_pack

```

10) Define do\_unpack() method which does the unpacking operation.

```

function void do_unpack(ovm_packer packer);

```

```

int sz;
super.do_pack(packer);

da = packer.unpack_field_int($bits(da));
sa = packer.unpack_field_int($bits(sa));
length = packer.unpack_field_int($bits(length));

data.delete();
data = new[length];
foreach(data[i])
    data[i] = packer.unpack_field_int(8);
fcs = packer.unpack_field_int($bits(fcs));
endfunction : do_unpack

```

### Packet class source code

```

`ifndef GUARD_PACKET
`define GUARD_PACKET

`include "ovm.svh"
import ovm_pkg::*;

//Define the enumerated types for packet types
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;

class Packet extends ovm_sequence_item;

    rand fcs_kind_t    fcs_kind;

    rand bit [7:0] length;
    rand bit [7:0] da;
    rand bit [7:0] sa;
    rand bit [7:0] data[];
    rand byte fcs;

    constraint payload_size_c { data.size inside { [1 : 6]}; }

    constraint length_c { length == data.size; }

    function new(string name = "");

```

```

super.new(name);
endfunction : new

function void post_randomize();
    if(fcs_kind == GOOD_FCS)
        fcs = 8'b0;
    else
        fcs = 8'b1;
    fcs = cal_fcs();
endfunction : post_randomize

<:/// method to calculate the fcs /////
virtual function byte cal_fcs;
    integer i;
    byte result ;
    result = 0;
    result = result ^ da;
    result = result ^ sa;
    result = result ^ length;
    for (i = 0;i< data.size;i++)
        result = result ^ data[i];
    result = fcs ^ result;
    return result;
endfunction : cal_fcs

`ovm_object_utils_begin(Packet)
`ovm_field_int(da, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_int(sa, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_int(length, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_array_int(data, OVM_ALL_ON|OVM_NOPACK)
`ovm_field_int(fcs, OVM_ALL_ON|OVM_NOPACK)
`ovm_object_utils_end

function void do_pack(ovm_packer packer);
    super.do_pack(packer);
    packer.pack_field_int(da,$bits(da));
    packer.pack_field_int(sa,$bits(sa));
    packer.pack_field_int(length,$bits(length));
    foreach(data[i])
        packer.pack_field_int(data[i],8);

```

```

    packer.pack_field_int(fcs,$bits(fcs));
endfunction : do_pack

function void do_unpack(ovm_packer packer);
    int sz;
    super.do_pack(packer);

    da = packer.unpack_field_int($bits(da));
    sa = packer.unpack_field_int($bits(sa));
    length = packer.unpack_field_int($bits(length));

    data.delete();
    data = new[length];
    foreach(data[i])
        data[i] = packer.unpack_field_int(8);
    fcs = packer.unpack_field_int($bits(fcs));
endfunction : do_unpack

endclass : Packet

```

### Test The Transaction Implementation

Now we will write a small logic to test our packet implantation. This module is not used in normal verification.

Define a module and take the instance of packet class. Randomize the packet and call the print method to analyze the generation. Then pack the packet in to bytes and then unpack bytes and then call compare method to check all the method implementation.

1) Declare Packet objects and dynamic arrays.

```

Packet pkt1 = new("pkt1");
Packet pkt2 = new("pkt2");
byte unsigned pkdbytes[];

```

2) In a initial block, randomize the packet, pack the packet in to pkdbytes and then unpack it and compare the packets.

```

if(pkt1.randomize)
begin
    $display(" Randomization Sucessesfull.");

```

```

pkt1.print();
ovm_default_packer.use_metadata = 1;
void'(pkt1.pack_bytes(pkdbytes));
$display("Size of pkd bits %d",pkdbytes.size());
pkt2.unpack_bytes(pkdbytes);
pkt2.print();
if(pkt2.compare(pkt1))
    $display(" Packing,Unpacking and compare worked");
else
    $display(" *** Something went wrong in Packing or Unpacking or compare *** \n \n");

```

### Logic to test the transaction implementation

**module** test;

```

Packet pkt1 = new("pkt1");
Packet pkt2 = new("pkt2");
byte unsigned pkdbytes[];

initial
repeat(10)
    if(pkt1.randomize)
        begin
            $display(" Randomization Successesfull.");
            pkt1.print();
            ovm_default_packer.use_metadata = 1;
            void'(pkt1.pack_bytes(pkdbytes));
            $display("Size of pkd bits %d",pkdbytes.size());
            pkt2.unpack_bytes(pkdbytes);
            pkt2.print();
            if(pkt2.compare(pkt1))
                $display(" Packing,Unpacking and compare worked");
            else
                $display(" *** Something went wrong in Packing or Unpacking or compare *** \n \n");
        end
        else
            $display(" *** Randomization Failed ***");

```

**endmodule**

## Download the Source Code

[ovm\\_switch\\_4.tar](#)

[Browse the code in ovm\\_switch\\_4.tar](#)

## Command to run the simulation

```
vcs -sverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv -R Packet.sv  
qverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv -R Packet.sv
```

## Log report after simulation

Randomization Sucessesfull.

Name	Type	Size	Value
pkt1	Packet	-	pkt1@3
--da	integral	8	'ha5
--sa	integral	8	'ha1
--length	integral	8	'h6
--data	da(integral)	6	-
---[0]	integral	8	'h58
---[1]	integral	8	'h60
---[2]	integral	8	'h34
---[3]	integral	8	'hdd
---[4]	integral	8	'h9
---[5]	integral	8	'haf
--fcs	integral	8	'h75

Size of pkd bits 10

Name	Type	Size	Value
pkt2	Packet	-	pkt2@5
--da	integral	8	'ha5
--sa	integral	8	'ha1
--length	integral	8	'h6
--data	da(integral)	6	-
---[0]	integral	8	'h58
---[1]	integral	8	'h60
---[2]	integral	8	'h34

```

----[3]      integral     8      'hdd
----[4]      integral     8      'h9
----[5]      integral     8      'haf
--fcs       integral     8      'h75
-----
```

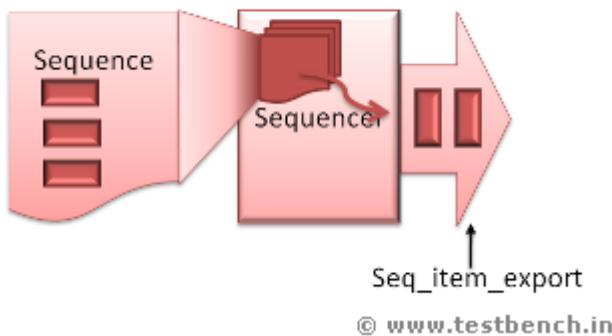
Packing,Unpacking and compare worked

## **PHASE 5 SEQUENCER N SEQUENCE**

In this phase we will develop Sequence and Sequencer.

A sequence is series of transaction and sequencer is used to for controlling the flow of transaction generation.

A sequence of transaction (which we already developed in previous phase) is defined by extending ovm\_sequence class. ovm\_sequencer does the generation of this sequence of transaction, ovm\_driver takes the transaction from Sequencer and processes the packet/ drives to other component or to DUT.



© www.testbench.in

## **Sequencer**

A Sequencer is defined by extending ovm\_sequencer. ovm\_sequencer has a port seq\_item\_export which is used to connect to ovm\_driver for transaction transfer.

- 1) Define a sequencer by extending ovm\_sequence.

```
'ifndef GUARD_SEQUENCER
`define GUARD_SEQUENCER
```

```
class Sequencer extends ovm_sequencer #(Packet);
```

```
endclass : Sequencer
```

```
`endif
```

2) We need Device port address, which are in configuration class. So declare a configuration class object.

```
Configuration cfg;
```

3) Declare Sequencer utility macros.

```
`ovm_sequencer_utils(Sequencer)
```

4) Define the constructor.

```
function new (string name, ovm_component parent);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(Packet)
endfunction : new
```

5) In `end_of_elaboration()` method, using `get_config_object()`, get the configuration object which will be passed from testcase.

`get_config_object()` returns object of type `ovm_object`, so using a temporary `ovm_object` and cast it to configuration object.

```
function void end_of_elaboration();
    ovm_object tmp;
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
endfunction
```

### Sequencer source code

```
`ifndef GUARD_SEQUENCER
`define GUARD_SEQUENCER
```

```
class Sequencer extends ovm_sequencer #(Packet);
```

```
Configuration cfg;
```

```

`ovm_sequencer_utils(Sequencer)

function new (string name, ovm_component parent);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(Packet)
endfunction : new

function void end_of_elaboration();
    ovm_object tmp;
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
endfunction

endclass : Sequencer

`endif
Sequence

```

A sequence is defined by extending ovm\_sequence class. This sequence of transactions should be defined in budy() method of ovm\_sequence class. OVM has macros and methods to define the transaction types. We will use macros in this example.

You can define as many sequences as you want. We will define 2 sequences.

1) Define sequence by extending

```

class Seq_device0_and_device1 extends ovm_sequence #(Packet);

endclass: Seq_device0_and_device1

```

2) Define constructor method.

```

function new(string name = "Seq_do");
    super.new(name);
endfunction : new

```

3) Declare utilities macro. With this macro, this sequence is tied to Sequencer.

```
`ovm_sequence_utils(Seq_device0_and_device1, Sequencer)
```

4) The algorithm for the transaction should be defined in body() method of the sequence. In this sequence we will define the algorithm such that alternate transactions for device port 0 and 1 are generated.

The device addresses are available in configuration object which is in sequencer. Every sequence has a handle to its sequence through p\_sequencer. Using p\_sequencer handle, access the device address.

```
virtual task body();
  forever begin
    `ovm_do_with(item, {da == p_sequencer.cfg.device0_add;});
    `ovm_do_with(item, {da == p_sequencer.cfg.device1_add;});
  end
endtask : body
```

#### Sequence Source Code

```
class Seq_device0_and_device1 extends ovm_sequence #(Packet);
  function new(string name = "Seq_device0_and_device1");
    super.new(name);
  endfunction : new
  Packet item;
  `ovm_sequence_utils(Seq_device0_and_device1, Sequencer)
  virtual task body();
    forever begin
      `ovm_do_with(item, {da == p_sequencer.cfg.device0_add;});
      `ovm_do_with(item, {da == p_sequencer.cfg.device1_add;});
    end
  endtask : body
endclass :Seq_device0_and_device1
```

#### One more Sequence

```
class Seq_constant_length extends ovm_sequence #(Packet);
  function new(string name = "Seq_constant_length");
    super.new(name);
  endfunction : new
  Packet item;
  `ovm_sequence_utils(Seq_constant_length, Sequencer)
  virtual task body();
    forever begin
      `ovm_do_with(item, {length == 10;da == p_sequencer.cfg.device0_add;});
    end
  endtask : body
endclass :Seq_constant_length
```

```
endtask : body  
endclass : Seq_constant_length
```

[Download the Source Code](#)

[ovm\\_switch\\_5.tar](#)

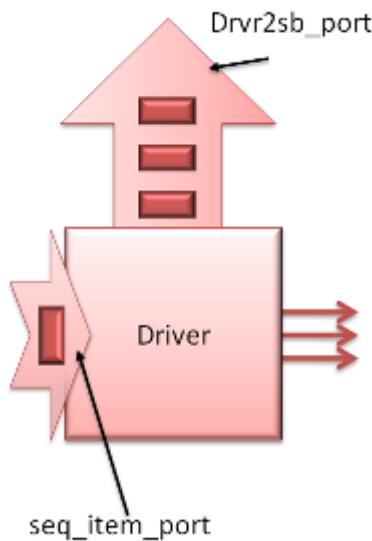
[Browse the code in ovm\\_switch\\_5.tar](#)

## **PHASE 6 DRIVER**

### **Driver**

In this phase we will develop the driver. Driver is defined by extending ovm\_driver. Driver takes the transaction from the sequencer using seq\_item\_port. This transaction will be driven to DUT as per the interface specification. After driving the transaction to DUT, it sends the transaction to scoreboard using ovm\_analysis\_port.

In driver class, we will also define task for resetting DUT and configuring the DUT. After completing the driver class implementation, we will instantiate it in environment class and connect the sequencer to it. We will also update the test case and run the simulation to check the implementation which we did till now.



© www.testbench.in

1) Define the driver class by extending ovm\_driver;

```
`ifndef GUARD_DRIVER  
`define GUARD_DRIVER  
class Driver extends ovm_driver #(Packet);  
endclass : Driver
```

2) Create a handle to configuration object. Using this object we can get DUT interfaces and DUT port addresses.

Configuration cfg;

3) Declare input and memory interfaces

```
virtual input_interface.IP input_intf;
virtual mem_interface.MEM mem_intf;
```

4) Declare ovm\_analysis\_port which is used to send packets to scoreboard.

```
ovm_analysis_port #(Packet) Drvr2Sb_port;
```

5) Declare component utilities macro.

```
`ovm_component_utils(Driver)
```

6) Define the constructor method. Pass the parent object to super class.

```
function new( string name = "", ovm_component parent = null) ;
    super.new( name , parent );
endfunction : new
```

7) In the build method and construct Drvr2Sb\_port object.

```
function void build();
    super.build();
    Drvr2Sb_port = new("Drvr2Sb_port", this);
endfunction : build
```

8) In the end\_of\_elaboration() method, get the configuration object using get\_config\_object and update the virtual interfaces.

```
function void end_of_elaboration();
    ovm_object tmp;
    super.end_of_elaboration();
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
    this.input_intf = cfg.input_intf;
    this.mem_intf = cfg.mem_intf;
endfunction : end_of_elaboration
```

9) Define the reset\_dut() method which will be used for resetting the DUT.

```
virtual task reset_dut();
    ovm_report_info(get_full_name(),"Start of reset_dut() method ",OVM_LOW);
    mem_intf.mem_data    <= 0;
    mem_intf.mem_add    <= 0;
    mem_intf.mem_en     <= 0;
    mem_intf.mem_rd_wr  <= 0;
    input_intf.data_in   <= 0;
    input_intf.data_status <= 0;

    input_intf.reset    <= 1;
    repeat (4) @ input_intf.clock;
```

```

input_intf.reset    <= 0;
ovm_report_info(get_full_name(),"End of reset_dut() method ",OVM_LOW);
endtask : reset_dut

10) Define the cfg_dut() method which does the configuration due port address.

virtual task cfg_dut();
    ovm_report_info(get_full_name(),"Start of cfg_dut() method ",OVM_LOW);
    mem_intf.mem_en <= 1;
    @(posedge mem_intf.clock);
    mem_intf.mem_rd_wr <= 1;

    @(posedge mem_intf.clock);
    mem_intf.mem_add <= 8'h0;
    mem_intf.mem_data <= cfg.device0_add;
    ovm_report_info(get_full_name(),
    $psprintf(" Port 0 Address %h ",cfg.device0_add),OVM_LOW);

    @(posedge mem_intf.clock);
    mem_intf.mem_add <= 8'h1;
    mem_intf.mem_data <= cfg.device1_add;
    ovm_report_info(get_full_name(),
    $psprintf(" Port 1 Address %h ",cfg.device1_add),OVM_LOW);

    @(posedge mem_intf.clock);
    mem_intf.mem_add <= 8'h2;
    mem_intf.mem_data <= cfg.device2_add;
    ovm_report_info(get_full_name(),
    $psprintf(" Port 2 Address %h ",cfg.device2_add),OVM_LOW);

    @(posedge mem_intf.clock);
    mem_intf.mem_add <= 8'h3;
    mem_intf.mem_data <= cfg.device3_add;
    ovm_report_info(get_full_name(),
    $psprintf(" Port 3 Address %h ",cfg.device3_add),OVM_LOW);

    @(posedge mem_intf.clock);
    mem_intf.mem_en  <=0;
    mem_intf.mem_rd_wr <= 0;
    mem_intf.mem_add  <= 0;
    mem_intf.mem_data <= 0;

```

```

    ovm_report_info(get_full_name(),"End of cfg_dut() method ",OVM_LOW);
endtask : cfg_dut

```

11) Define drive() method which will be used to drive the packet to DUT. In this method pack the packet fields using the pack\_bytes() method of the transaction and drive the packed data to DUT interface.

```

task drive(Packet pkt);
    byte unsigned bytes[];
    int pkt_len;
    pkt_len = pkt.pack_bytes(bytes);
    ovm_report_info(get_full_name(),"Driving packet ...",OVM_LOW);

    foreach(bytes[i])
    begin
        @(posedge input_intf.clock);
        input_intf.data_status <= 1 ;
        input_intf.data_in <= bytes[i];
    end

    @(posedge input_intf.clock);
    input_intf.data_status <= 0 ;
    input_intf.data_in <= 0;
    repeat(2) @(posedge input_intf.clock);
endtask : drive

```

12) Now we will use the above 3 defined methods and update the run() method of ovm\_driver. First call the reset\_dut() method and then cfg\_dut(). After completing the configuration, in a forever loop get the transaction from seq\_item\_port and send it DUT using drive() method and also to scoreboard using Drvr2SB\_port .

```

task run();
    Packet pkt;
    @(posedge input_intf.clock);
    reset_dut();
    cfg_dut();
    forever begin
        seq_item_port.get_next_item(pkt);
        Drvr2Sb_port.write(pkt);
        @(posedge input_intf.clock);

```

```

    drive(pkt);
    @(posedge input_intf.clock);
    seq_item_port.item_done();
end
endtask : run

```

### Driver class source code

```

`ifndef GUARD_DRIVER
`define GUARD_DRIVER
class Driver extends ovm_driver #(Packet);
    Configuration cfg;
    virtual input_interface.IP    input_intf;
    virtual mem_interface.MEM   mem_intf;

    ovm_analysis_port #(Packet) Drvr2Sb_port;

`ovm_component_utils(Driver)

function new( string name = "" , ovm_component parent = null ) ;
    super.new( name , parent );
endfunction : new

function void build();
    super.build();
    Drvr2Sb_port = new("Drvr2Sb", this);
endfunction : build

function void end_of_elaboration();
    ovm_object tmp;
    super.end_of_elaboration();
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
    this.input_intf = cfg.input_intf;
    this.mem_intf = cfg.mem_intf;
endfunction : end_of_elaboration

task run();
    Packet pkt;
    @(posedge input_intf.clock);
    reset_dut();

```

```

cfg_dut();
forever begin
    seq_item_port.get_next_item(pkt);
    Drvr2Sb_port.write(pkt);
    @(posedge input_intf.clock);
    drive(pkt);
    @(posedge input_intf.clock);
    seq_item_port.item_done();
end
endtask : run

virtual task reset_dut();
    ovm_report_info(get_full_name(),"Start of reset_dut() method ",OVM_LOW);
    mem_intf.mem_data    <= 0;
    mem_intf.mem_add     <= 0;
    mem_intf.mem_en      <= 0;
    mem_intf.mem_rd_wr   <= 0;
    input_intf.data_in   <= 0;
    input_intf.data_status <= 0;

    input_intf.reset     <= 1;
    repeat (4) @ input_intf.clock;
        input_intf.reset     <= 0;

    ovm_report_info(get_full_name(),"End of reset_dut() method ",OVM_LOW);
endtask : reset_dut

virtual task cfg_dut();
    ovm_report_info(get_full_name(),"Start of cfg_dut() method ",OVM_LOW);
    mem_intf.mem_en <= 1;
    @(posedge mem_intf.clock);
    mem_intf.mem_rd_wr <= 1;

    @(posedge mem_intf.clock);
    mem_intf.mem_add <= 8'h0;
    mem_intf.mem_data <= cfg.device0_add;
    ovm_report_info(get_full_name(),
    $psprintf(" Port 0 Address %h ",cfg.device0_add),OVM_LOW);

    @(posedge mem_intf.clock);

```

```

mem_intf.mem_add <= 8'h1;
mem_intf.mem_data <= cfg.device1_add;
ovm_report_info(get_full_name(),
$psprintf(" Port 1 Address %h ",cfg.device1_add),OVM_LOW);

@(posedge mem_intf.clock);
mem_intf.mem_add <= 8'h2;
mem_intf.mem_data <= cfg.device2_add;
ovm_report_info(get_full_name(),
$psprintf(" Port 2 Address %h ",cfg.device2_add),OVM_LOW);

@(posedge mem_intf.clock);
mem_intf.mem_add <= 8'h3;
mem_intf.mem_data <= cfg.device3_add;
ovm_report_info(get_full_name(),
$psprintf(" Port 3 Address %h ",cfg.device3_add),OVM_LOW);

@(posedge mem_intf.clock);
mem_intf.mem_en <=0;
mem_intf.mem_rd_wr <= 0;
mem_intf.mem_add <= 0;
mem_intf.mem_data <= 0;

ovm_report_info(get_full_name(),"End of cfg_dut() method ",OVM_LOW);
endtask : cfg_dut

task drive(Packet pkt);
  byte unsigned bytes[];
  int pkt_len;
  pkt_len = pkt.pack_bytes(bytes);
  ovm_report_info(get_full_name(),"Driving packet ...",OVM_LOW);

  foreach(bytes[i])
    begin
      @(posedge input_intf.clock);
      input_intf.data_status <= 1 ;
      input_intf.data_in <= bytes[i];
    end

    @(posedge input_intf.clock);

```

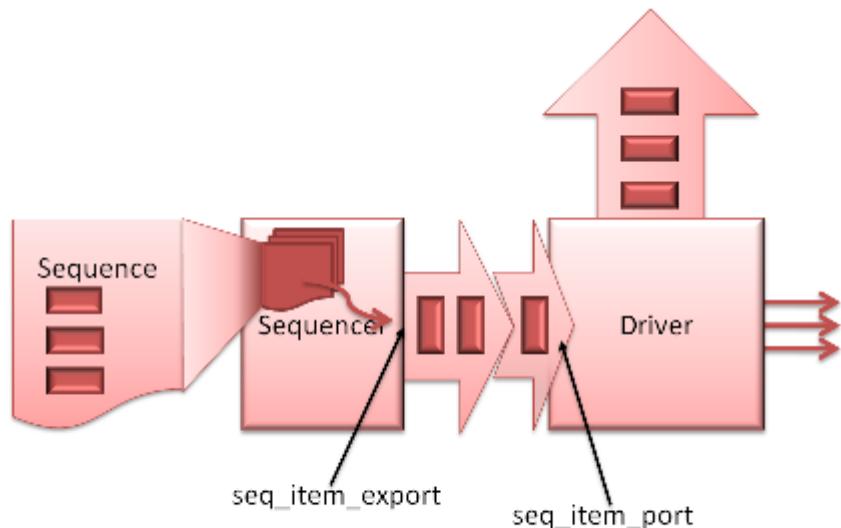
```

input_intf.data_status <= 0 ;
input_intf.data_in <= 0;
repeat(2) @(posedge input_intf.clock);
endtask : drive
endclass : Driver
`endif

```

### Environment Updates

We will take the instance of Sequencer and Driver and connect them in Environment class.



© www.testbench.in

- 1) Declare handles to Driver and Sequencer.

```

Sequencer Seqncr;
Driver Drvr;

```

- 2) In build method, construct Seqncr and Drvr object using create() method.

```

Drvr = Driver::type_id::create("Drvr",this);
Seqncr = Sequencer::type_id::create("Seqncr",this);

```

- 2) In connect() method connect the sequencer seq\_item\_port to drivers seq\_item\_export.

```
Drvr.seq_item_port.connect(Seqncr.seq_item_export);
```

### Environment class code

```
`ifndef GUARD_ENV
`define GUARD_ENV
class Environment extends ovm_env;
  `ovm_component_utils(Environment)

  Sequencer Seqncr;
  Driver Drvr;
  function new(string name , ovm_component parent = null);
    super.new(name, parent);
  endfunction: new

  function void build();
    super.build();
    ovm_report_info(get_full_name(),"START of build ",OVM_LOW);

    Drvr = Driver::type_id::create("Drvr",this);
    Seqncr = Sequencer::type_id::create("Seqncr",this);

    ovm_report_info(get_full_name(),"END of build ",OVM_LOW);
  endfunction

  function void connect();
    super.connect();
    ovm_report_info(get_full_name(),"START of connect ",OVM_LOW);

    Drvr.seq_item_port.connect(Seqncr.seq_item_export);

    ovm_report_info(get_full_name(),"END of connect ",OVM_LOW);
  endfunction

endclass : Environment
`endif
```

### Testcase Updates

We will update the testcase and run the simulation.

- 1)In the build() method, update the configuration address in the configuration object which in top module.

```
function void build();
  super.build();
```

```
cfg.device0_add = 0;
cfg.device1_add = 1;
cfg.device2_add = 2;
cfg.device3_add = 3;
```

2) In the build() method itself, using set\_config\_object , configure the configuration object with the one which is in top module.

with this, the configuration object in Sequencer and Driver will be pointing to the one which in top module.

```
set_config_object("t_env.*","Configuration",cfg);
```

3) In the build method, using set\_config\_string, configure the default\_sequence of the sequencer to use the sequence which we defined.

```
set_config_string("*.Seqncr", "default_sequence", "Seq_device0_and_device1");
```

4) Set the sequencer count value to 2 .

```
set_config_int("*.Seqncr", "count",2);
```

5) Update the run() method to print the Sequencer details.

```
t_env.Seqncr.print();
```

### Testcase code

```
class test1 extends ovm_test;
`ovm_component_utils(test1)

Environment t_env;

function new (string name="test1", ovm_component parent=null);
    super.new (name, parent);
    t_env = new("t_env",this);
endfunction : new

function void build();
```

```

super.build();

cfg.device0_add = 0;
cfg.device1_add = 1;
cfg.device2_add = 2;
cfg.device3_add = 3;

set_config_object("t_env.*","Configuration",cfg);
set_config_string("*.Seqncr", "default_sequence", "Seq_device0_and_device1");
set_config_int("*.Seqncr", "count",2);

endfunction

task run ();
    t_env.Seqncr.print();

    #1000;
    global_stop_request();
endtask : run

endclass : test1

```

### Download the source code

[ovm\\_switch\\_6.tar](#)

[Browse the code in ovm\\_switch\\_6.tar](#)

### Command to run the simulation

```

vcs -sverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1
qverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1

```

### Log report after simulation

OVM\_INFO @ 0 [RNTST] Running test test1...

OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] START of build  
OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] END of build  
OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] START of connect  
OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] END of connect

---

Name	Type	Size	Value
Seqncr	Sequencer	-	Seqncr@14
--rsp_export	ovm_analysis_export	-	rsp_export@16
--seq_item_export	ovm_seq_item_pull_+	-	seq_item_export@40
--default_sequence	string	19	ovm_random_sequence
--count	integral	32	-1
--max_random_count	integral	32	'd10
--sequences	array	5	-
----[0]	string	19	ovm_random_sequence
----[1]	string	23	ovm_exhaustive_sequ+
----[2]	string	19	ovm_simple_sequence
----[3]	string	23	Seq_device0_and_dev+
----[4]	string	19	Seq_constant_length
--max_random_depth	integral	32	'd4
--num_last_reqs	integral	32	'd1
--num_last_rsp	integral	32	'd1

---

OVM\_INFO @ 30: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Start of reset\_dut() method  
OVM\_INFO @ 70: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
End of reset\_dut() method  
OVM\_INFO @ 70: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Start of cfg\_dut() method  
OVM\_INFO @ 110: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 0 Address 00  
OVM\_INFO @ 130: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 1 Address 01  
OVM\_INFO @ 150: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 2 Address 02  
OVM\_INFO @ 170: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 3 Address 03  
OVM\_INFO @ 190: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
End of cfg\_dut() method  
OVM\_INFO @ 210: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]

```

Driving packet ...
OVM_INFO @ 590: ovm_test_top.t_env.Drvr [ovm_test_top.t_env.Drvr]
    Driving packet ...
OVM_INFO @ 970: ovm_test_top.t_env.Drvr [ovm_test_top.t_env.Drvr]
    Driving packet ...

--- OVM Report Summary ---

** Report counts by severity
```

```

OVM_INFO : 16
OVM_WARNING : 0
OVM_ERROR : 0
OVM_FATAL : 0
```

## **PHASE 7 RECEIVER**

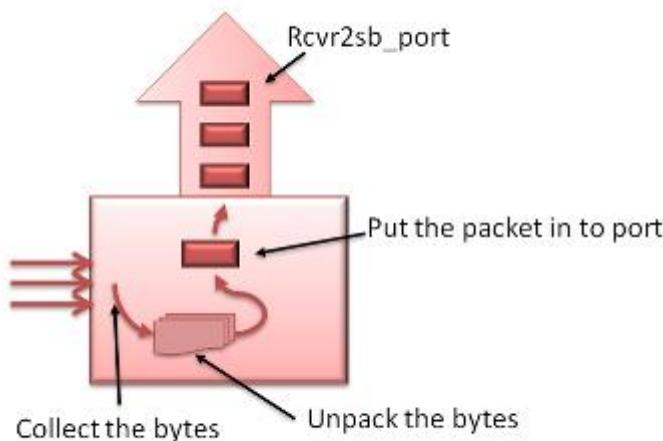
In this phase, we will write a receiver and use the receiver in environment class to collect the packets coming from the switch output\_interface.

### **Receiver**

Receiver collects the data bytes from the interface signal. And then unpacks the bytes in to packet using unpack\_bytes method and pushes it into Rcvr2Sb\_port for score boarding.

Receiver class is written in Receiver.sv file.

Receiver class is defined by extending ovm\_component class. It will drive the received transaction to scoreboard using ovm\_analysis\_port.



1) Define Receiver class by extending ovm\_component.

```
`ifndef GUARD_RECEIVER
`define GUARD_RECEIVER
class Receiver extends ovm_component;
endclass : Receiver
`endif
```

2) Declare configuration class object.

```
Configuration cfg;
```

3) Declare an integer to hold the receiver number.

```
integer id;
```

4) Declare a virtual interface of dut out put side.

```
virtual output_interface#OP output_intf;
```

5) Declare analysis port which is used by receiver to send the received transaction to scoreboard.

```
ovm_analysis_port #(Packet) Rcvr2Sb_port;
```

6) Declare the utility macro. This utility macro provides the implementation of creat() and get\_type\_name() methods.

```
`ovm_component_utils(Receiver)
```

7) Define the constructor.

```
function new (string name, ovm_component parent);
    super.new(name, parent);
endfunction : new
```

8) Define the build method and construct the Rcvr2Sb\_port.

```
function void build();
    super.build();
    Rcvr2Sb_port = new("Rcvr2Sb", this);
endfunction : build
```

9) In the end\_of\_elaboration() method, get the configuration object using get\_config\_object and update the virtual interfaces.

```
function void end_of_elaboration();
    ovm_object tmp;
    super.end_of_elaboration();
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
    output_intf = cfg.output_intf[id];
endfunction : end_of_elaboration
```

10) Define the run() method. This method collects the packets from the DUT output interface and unpacks it into high level transaction using transactions unpack\_bytes() method.

```
virtual task run();
    bit [7:0] bytes[];
```

```

Packet pkt;
fork
  forever
    begin
      repeat(2) @@(posedge output_intf.clock);
      wait(output_intf.ready)
      output_intf.read <= 1;

      repeat(2) @@(posedge output_intf.clock);
      while (output_intf.ready)
      begin
        bytes = new[bytes.size + 1](bytes);
        bytes[bytes.size - 1] = output_intf.data_out;
        @(posedge output_intf.clock);
      end

      output_intf.read <= 0;
      @(posedge output_intf.clock);
      ovm_report_info(get_full_name(),"Received packet ...",OVM_LOW);
      pkt = new();
      pkt.unpack_bytes(bytes);
      Rcvr2Sb_port.write(pkt);
      bytes.delete();
    end
  join

```

**endtask** : run

#### Receiver class source code

```

`ifndef GUARD_RECEIVER
`define GUARD_RECEIVER
class Receiver extends ovm_component;
  virtual output_interface.OP output_intf;
  Configuration cfg;
  integer id;

  ovm_analysis_port #(Packet) Rcvr2Sb_port;
  `ovm_component_utils(Receiver)
  function new (string name, ovm_component parent);
    super.new(name, parent);
  endfunction : new
  function void build();

```

```

super.build();
Rcvr2Sb_port = new("Rcvr2Sb", this);
endfunction : build

function void end_of_elaboration();
    ovm_object tmp;
    super.end_of_elaboration();
    assert(get_config_object("Configuration",tmp));
    $cast(cfg,tmp);
    output_intf = cfg.output_intf[id];
endfunction : end_of_elaboration

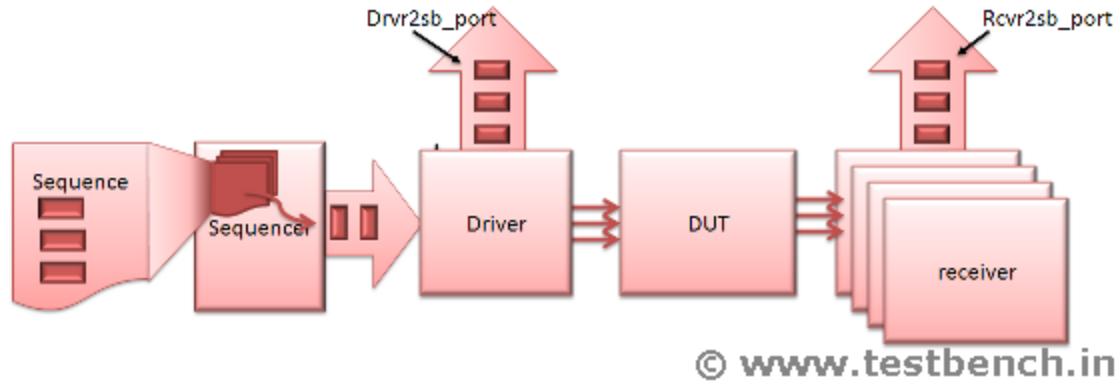
virtual task run();
    bit [7:0] bytes[];
    Packet pkt;
fork
    forever
    begin
        repeat(2) @(posedge output_intf.clock);
        wait(output_intf.ready)
        output_intf.read <= 1;
        repeat(2) @(posedge output_intf.clock);
        while (output_intf.ready)
        begin
            bytes = new[bytes.size + 1](bytes);
            bytes[bytes.size - 1] = output_intf.data_out;
            @(posedge output_intf.clock);
        end
        output_intf.read <= 0;
        @(posedge output_intf.clock);
        ovm_report_info(get_full_name(),"Received packet ...",OVM_LOW);
        pkt = new();
        pkt.unpack_bytes(bytes);
        Rcvr2Sb_port.write(pkt);
        bytes.delete();
    end
    join
endtask : run

endclass : Receiver

```

## Environment Class Updates

We will update the Environment class and take instance of receiver and run the testcase.



- 1) Declare 4 receivers.

Receiver Rcvr[4];

- 2) In the build() method construct the Receivers using create() methods. Also update the id variable of the receiver object.

```
foreach(Rcvr[i]) begin
    Rcvr[i] = Receiver::type_id::create($psprintf("Rcvr%0d",i),this);
    Rcvr[i].id = i;
end
```

### Environment class source code

```
`ifndef GUARD_ENV
`define GUARD_ENV
class Environment extends ovm_env;

    `ovm_component_utils(Environment)

    Sequencer Seqncr;
    Driver Drvr;

    Receiver Rcvr[4];
    function new(string name , ovm_component parent = null);
        super.new(name, parent);
    endfunction: new
```

```

function void build();
super.build();
ovm_report_info(get_full_name(),"START of build ",OVM_LOW);
Drvrv = Driver::type_id::create("Drvrv",this);
Seqncr = Sequencer::type_id::create("Seqncr",this);

foreach(Rcvr[i]) begin
    Rcvr[i] = Receiver::type_id::create($psprintf("Rcvr%0d",i),this);
    Rcvr[i].id = i;
end

    ovm_report_info(get_full_name(),"END of build ",OVM_LOW);
endfunction

function void connect();
super.connect();
ovm_report_info(get_full_name(),"START of connect ",OVM_LOW);
Drvrv.seq_item_port.connect(Seqncr.seq_item_export);
ovm_report_info(get_full_name(),"END of connect ",OVM_LOW);
endfunction

endclass : Environment
`endif

```

### Download the Source Code

[ovm\\_switch\\_7.tar](#)

[Browse the code in ovm\\_switch\\_7.tar](#)

### Command to run the simulation

```

vcs -sverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1
qverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1

```

### Log report after simulation

OVM\_INFO @ 0 [RNTST] Running test test1...

OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] START of build  
OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] END of build  
OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] START of connect  
OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] END of connect

---

Name	Type	Size	Value
Seqncr	Sequencer	-	Seqncr@14
--rsp_export	ovm_analysis_export	-	rsp_export@16
--seq_item_export	ovm_seq_item_pull_+	-	seq_item_export@40
--default_sequence	string	19	ovm_random_sequence
--count	integral	32	-1
--max_random_count	integral	32	'd10
--sequences	array	5	-
----[0]	string	19	ovm_random_sequence
----[1]	string	23	ovm_exhaustive_sequ+
----[2]	string	19	ovm_simple_sequence
----[3]	string	23	Seq_device0_and_dev+
----[4]	string	19	Seq_constant_length
--max_random_depth	integral	32	'd4
--num_last_reqs	integral	32	'd1
--num_last_rsp	integral	32	'd1

---

OVM\_INFO @ 30: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Start of reset\_dut() method  
OVM\_INFO @ 70: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
End of reset\_dut() method  
OVM\_INFO @ 70: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Start of cfg\_dut() method  
OVM\_INFO @ 110: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 0 Address 00  
OVM\_INFO @ 130: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 1 Address 01  
OVM\_INFO @ 150: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 2 Address 02  
OVM\_INFO @ 170: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
Port 3 Address 03  
OVM\_INFO @ 190: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
End of cfg\_dut() method  
OVM\_INFO @ 210: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]

```
    Driving packet ...
OVM_INFO @ 590: ovm_test_top.t_env.Drvr [ovm_test_top.t_env.Drvr]
    Driving packet ...
OVM_INFO @ 610: ovm_test_top.t_env.Rcvr0 [ovm_test_top.t_env.Rcvr0]
    Received packet ...
OVM_INFO @ 970: ovm_test_top.t_env.Drvr [ovm_test_top.t_env.Drvr]
    Driving packet ...
OVM_INFO @ 990: ovm_test_top.t_env.Rcvr0 [ovm_test_top.t_env.Rcvr0]
    Received packet ...
```

--- OVM Report Summary ---

\*\* Report counts by severity

```
OVM_INFO : 18
OVM_WARNING : 0
OVM_ERROR : 0
OVM_FATAL : 0
```

## **PHASE 8 SCOREBOARD**

In this phase we will see the scoreboard implementation.

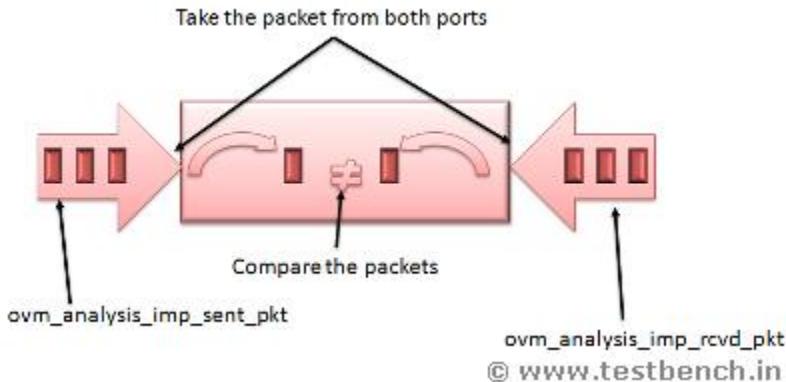
### **Scoreboard**

Scoreboard is implemented by extending ovm\_scoreboard. For our requirement, we can use ovm\_in\_order\_comparator, but we will see develop our own scoreboard by extending ovm\_scoreboard. Scoreboard has 2 analysis imports. One is used to for getting the packets from the driver and other from the receiver. Then the packets are compared and if they don't match, then error is asserted. For comparison, compare () method of the Packet class is used.

Implement the scoreboard in file Scoreboard.sv.

Steps to create a scoreboard:

- 1) Using macro `ovm\_analysis\_imp\_decl(<\_portname>), to declare ovm\_analysis\_imp\_<\_portname> class.
- 2) The above macro, creates write\_<\_portname>(). This method has to be defined as per our requirements.



1) Declare a scoreboard by extending ovm\_scoreboard class.

```
class Scoreboard extends ovm_scoreboard;
endclass : Scoreboard
```

2) We need 2 import, one for expected packet which is sent by driver and received packet which is coming from receiver.

Declare 2 imports using `ovm\_analysis\_imp\_decl macros.

```
`ovm_analysis_imp_decl(_rcvd_pkt)
`ovm_analysis_imp_decl(_sent_pkt)
```

3) Declare the utility macro.

```
`ovm_component_utils(Scoreboard)
```

4) Declare a queue which stores the expected packets.

```
Packet exp_QUE[$];
```

5) Declare imports for getting expected packets and received packets.

```
ovm_analysis_imp_rcvd_pkt #(Packet,Scoreboard) Rcvr2Sb_port;
ovm_analysis_imp_sent_pkt #(Packet,Scoreboard) Drvr2Sb_port;
```

6) In the constructor, create objects for the above two declared imports.

```
function new(string name, ovm_component parent);
    super.new(name, parent);
    Rcvr2Sb_port = new("Rcvr2Sb", this);
```

```

    Drvr2Sb_port = new("Drvr2Sb", this);
endfunction : new

```

- 7) Define write\_sent\_pkt() method which was created by macro  
`ovm\_analysis\_imp\_decl(\_sent\_pkt).

In this method, store the received packet in the expected queue.

```

virtual function void write_sent_pkt(input Packet pkt);
    exp_que.push_back(pkt);
endfunction : write_sent_pkt

```

- 8) Define write\_rcvd\_pkt() method which was created by macro  
`ovm\_analysis\_imp\_decl(\_rcvd\_pkt)

In this method, get the transaction from the expected queue and compare.

```

virtual function void write_rcvd_pkt(input Packet pkt);
    Packet exp_pkt;
    pkt.print();
    if(exp_que.size())
        begin
            exp_pkt = exp_que.pop_front();
            exp_pkt.print();
            if( pkt.compare(exp_pkt))
                ovm_report_info(get_type_name(),
                    $psprintf("Sent packet and received packet matched"), OVM_LOW);
            else
                ovm_report_error(get_type_name(),
                    $psprintf("Sent packet and received packet mismatched"), OVM_LOW);
        end
        else
            ovm_report_error(get_type_name(),
                $psprintf("No more packets in the expected queue to compare"), OVM_LOW);
endfunction : write_rcvd_pkt

```

- 9) Define the report() method to print the Scoreboard information.

```

virtual function void report();
    ovm_report_info(get_type_name(),
        $psprintf("Scoreboard Report %s", this.sprint()), OVM_LOW);
endfunction : report

```

### Complete Scoreboard Code

```
`ifndef GUARD_SCOREBOARD
`define GUARD_SCOREBOARD

`ovm_analysis_imp_decl(_rcvd_pkt)
`ovm_analysis_imp_decl(_sent_pkt)

class Scoreboard extends ovm_scoreboard;
  `ovm_component_utils(Scoreboard)

  Packet exp_que[$];

  ovm_analysis_imp_rcvd_pkt #(Packet,Scoreboard) Rcvr2Sb_port;
  ovm_analysis_imp_sent_pkt #(Packet,Scoreboard) Drvr2Sb_port;

  function new(string name, ovm_component parent);
    super.new(name, parent);
    Rcvr2Sb_port = new("Rcvr2Sb", this);
    Drvr2Sb_port = new("Drvr2Sb", this);
  endfunction : new

  virtual function void write_rcvd_pkt(input Packet pkt);
    Packet exp_pkt;
    pkt.print();

    if(exp_que.size())
      begin
        exp_pkt = exp_que.pop_front();
        exp_pkt.print();
        if( pkt.compare(exp_pkt))
          ovm_report_info(get_type_name(),
            $psprintf("Sent packet and received packet matched"), OVM_LOW);
        else
          ovm_report_error(get_type_name(),
            $psprintf("Sent packet and received packet mismatched"), OVM_LOW);
      end
    else
      ovm_report_error(get_type_name(),
        $psprintf("No more packets to in the expected queue to compare"), OVM_LOW);
  end
end
```

```

endfunction : write_rcvd_pkt

virtual function void write_sent_pkt(input Packet pkt);
    exp_que.push_back(pkt);
endfunction : write_sent_pkt

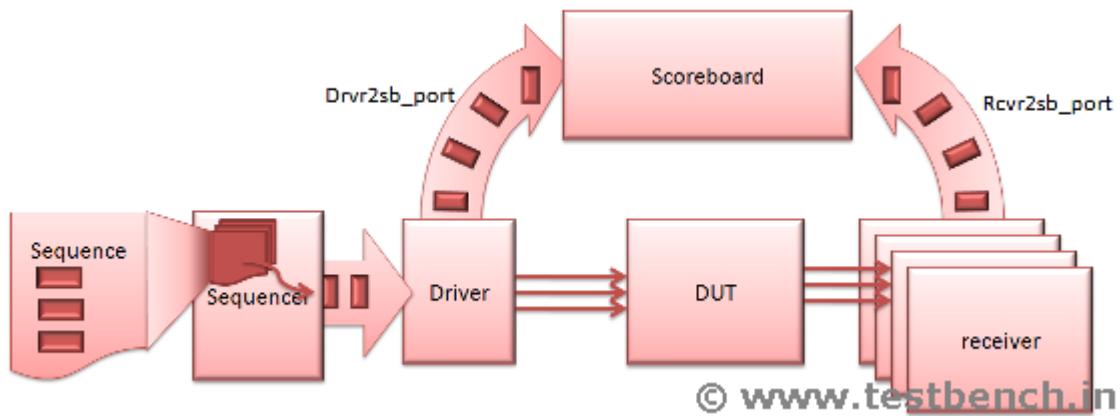
virtual function void report();
    ovm_report_info(get_type_name(),
        $psprintf("Scoreboard Report %s", this.sprint()), OVM_LOW);
endfunction : report

endclass : Scoreboard
`endif

```

### Environment Class Updates

We will take the instance of scoreboard in the environment and connect its ports to driver and receiver ports.



- 1) Declare scoreboard object.

```
Scoreboard Sbd;
```

- 2) Construct the scoreboard object using create() method in build() method.

```
Sbd = Scoreboard::type_id::create("Sbd",this);
```

- 3) In connect() method, connect the driver and receiver ports to scoreboard.

```

Drvrv.Drvr2Sb_port.connect(Sbd.Drvr2Sb_port);

Rcvr[0].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);
Rcvr[1].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);
Rcvr[2].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);
Rcvr[3].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);

```

### Environemnt class code

```

`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends ovm_env;

    `ovm_component_utils(Environment)

    Sequencer Seqncr;
    Driver Drvr;
    Receiver Rcvr[4];

    Scoreboard Sbd;

    function new(string name , ovm_component parent = null);
        super.new(name, parent);
    endfunction: new

    function void build();
        super.build();
        ovm_report_info(get_full_name(),"START of build ",OVM_LOW);

        Drvr = Driver::type_id::create("Drvr",this);
        Seqncr = Sequencer::type_id::create("Seqncr",this);

        foreach(Rcvr[i]) begin
            Rcvr[i] = Receiver::type_id::create($psprintf("Rcvr%0d",i),this);
            Rcvr[i].id = i;
        end

        Sbd = Scoreboard::type_id::create("Sbd",this);
    endfunction: build
endclass: Environment

```

```

    ovm_report_info(get_full_name(),"END of build ",OVM_LOW);
endfunction

function void connect();
    super.connect();
    ovm_report_info(get_full_name(),"START of connect ",OVM_LOW);

    Drvr.seq_item_port.connect(Seqncr.seq_item_export);

    Drvr.Drvr2Sb_port.connect(Sbd.Drvr2Sb_port);

    Rcvr[0].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);
    Rcvr[1].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);
    Rcvr[2].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);
    Rcvr[3].Rcvr2Sb_port.connect(Sbd.Rcvr2Sb_port);

    ovm_report_info(get_full_name(),"END of connect ",OVM_LOW);
endfunction

```

endclass : Environment  
`endif

### Download the Source Code

[ovm\\_switch\\_8.tar](#)

[Browse the code in ovm\\_switch\\_8.tar](#)

### Command to run the simulation

```
vcs -sverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1
qverilog +incdir+$OVM_HOME/src $OVM_HOME/src/ovm_pkg.sv +incdir+. rtl.sv
interface.sv top.sv -R +OVM_TESTNAME=test1
```

### Log report after simulation

```
OVM_INFO @ 0 [RNTST] Running test test1...
OVM_INFO @ 0: ovm_test_top.t_env [ovm_test_top.t_env] START of build
```

OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] END of build  
 OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] START of connect  
 OVM\_INFO @ 0: ovm\_test\_top.t\_env [ovm\_test\_top.t\_env] END of connect

---

Name	Type	Size	Value
<hr/>			
Seqncr	Sequencer	-	Seqncr@14
rsp_export	ovm_analysis_export	-	rsp_export@16
seq_item_export	ovm_seq_item_pull_+ -	-	seq_item_export@40
default_sequence	string	19	ovm_random_sequence
count	integral	32	-1
max_random_count	integral	32	'd10
sequences	array	5	-
[0]	string	19	ovm_random_sequence
[1]	string	23	ovm_exhaustive_sequ+
[2]	string	19	ovm_simple_sequence
[3]	string	23	Seq_device0_and_dev+
[4]	string	19	Seq_constant_length
max_random_depth	integral	32	'd4
num_last_reqs	integral	32	'd1
num_last_rsp	integral	32	'd1

---

OVM\_INFO @ 30: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 Start of reset\_dut() method  
 OVM\_INFO @ 70: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 End of reset\_dut() method  
 OVM\_INFO @ 70: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 Start of cfg\_dut() method  
 OVM\_INFO @ 110: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 Port 0 Address 00  
 OVM\_INFO @ 130: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 Port 1 Address 01  
 OVM\_INFO @ 150: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 Port 2 Address 02  
 OVM\_INFO @ 170: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 Port 3 Address 03  
 OVM\_INFO @ 190: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 End of cfg\_dut() method  
 OVM\_INFO @ 210: ovm\_test\_top.t\_env.Drvr [ovm\_test\_top.t\_env.Drvr]  
 Driving packet ...

```
OVM_INFO @ 590: ovm_test_top.t_env.Drvr [ovm_test_top.t_env.Drvr]
    Driving packet ...
OVM_INFO @ 610: ovm_test_top.t_env.Rcvr0 [ovm_test_top.t_env.Rcvr0]
    Received packet ...
OVM_INFO @ 610: ovm_test_top.t_env.Sbd [Scoreboard]
    Sent packet and received packet matched
OVM_INFO @ 970: ovm_test_top.t_env.Drvr [ovm_test_top.t_env.Drvr]
    Driving packet ...
OVM_INFO @ 990: ovm_test_top.t_env.Rcvr0 [ovm_test_top.t_env.Rcvr0]
    Received packet ...
OVM_INFO @ 990: ovm_test_top.t_env.Sbd [Scoreboard]
    Sent packet and received packet matched
OVM_INFO @ 1000: ovm_test_top.t_env.Sbd [Scoreboard]
    Scoreboard Report
```

---

Name	Type	Size	Value
<hr/>			
Sbd	Scoreboard	-	Sbd@52
Drvrv2Sb	ovm_analysis_imp_s+ -		Drvrv2Sb@56
Rcvrv2Sb	ovm_analysis_imp_r+ -		Rcvrv2Sb@54
<hr/>			

--- OVM Report Summary ---

```
** Report counts by severity
OVM_INFO : 21
OVM_WARNING : 0
OVM_ERROR : 0
OVM_FATAL : 0
```

## **INDEX** **INTRODUCTION**

### **SPECIFICATION**

- Switch Specification
- Packet Format
- Configuration
- Interface Specification

### **VERIFICATION PLAN**

- Overview
- Feature Extraction
- Stimulus Generation Plan
- Coverage Plan
- Verification Environment

### **PHASE 1 TOP**

- Interfaces
- Testcase
- Top Module
- Top Module Source Code **PHASE 2 ENVIRONMENT**
- Environment Class
- Run
- Environment Class Source Code

### **PHASE 3 RESET**

### **PHASE 4 PACKET**

- Packet Class Source Code
- Program Block Source Code

### **PHASE 5 GENERATOR**

- Environment Class Source Code

### **PHASE 6 DRIVER**

- Driver Class Source Code
- Environment Class Source Code

### **PHASE 7 RECEIVER**

- Receiver Class Source Code
- Environment Class Source Code

### **PHASE 8 SCOREBOARD**

- Scoreboard Class Source Code
- Source Code Of The Environment Class

### **PHASE 9 COVERAGE**

- Source Code Of Coverage Class

## **INTRODUCTION**

In this tutorial, we will verify the Switch RTL core using VMM in SystemVerilog. Following are the steps we follow to verify the Switch RTL core.

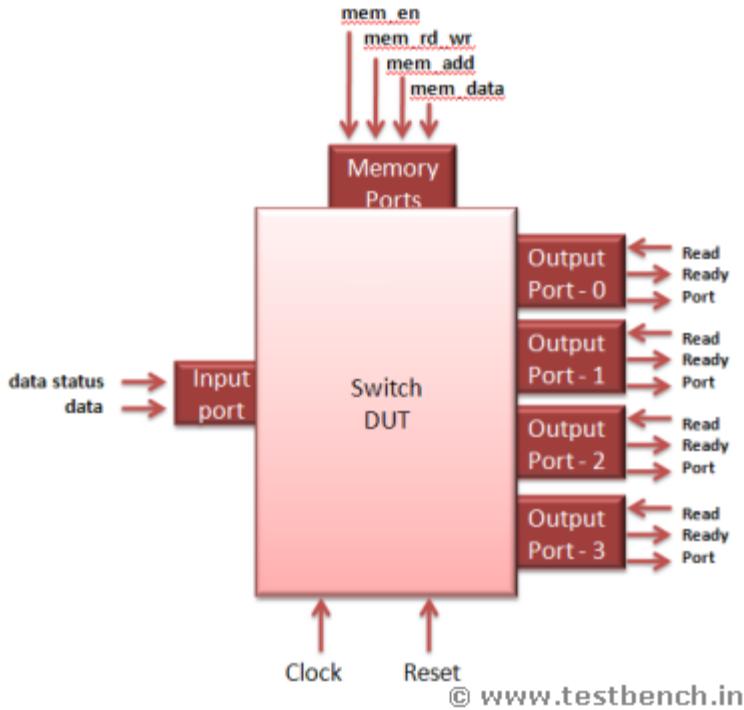
- 1) Understand the specification
- 2) Developing Verification Plan
- 3) Building the Verification Environment. We will build the Environment in Multiple phases, so it will be easy for you to learn step by step.
  - ⌚ Phase 1) We will develop the testcase and interfaces, and integrate them in these with the DUT in top module.
  - ⌚ Phase 2) We will Develop the Environment class.
  - ⌚ Phase 3) We will develop reset and configuration methods in Environment class. Then using these methods, we will reset the DUT and configure the port address.
  - ⌚ Phase 4) We will develop a packet class based on the stimulus plan. We will also write a small code to test the packet class implementation.
  - ⌚ Phase 5) We will create atomic generator in the environment class.
  - ⌚ Phase 6) We will develop a driver class. Packets are taken from the generator and sent to DUT using driver.
  - ⌚ Phase 7) We will develop receiver class. Receiver collects the packets coming from the output port of the DUT.
  - ⌚ Phase 8) We will develop scoreboard class which does the comparison of the expected packet with the actual packet received from the DUT.
  - ⌚ Phase 9) We will develop coverage class based on the coverage plan. Coverage is sampled using the driver callbacks.

## **SPECIFICATION**

### **Switch Specification:**

This is a simple switch. Switch is a packet based protocol. Switch drives the incoming packet which comes from the input port to output ports based on the address contained in the packet.

The switch has a one input port from which the packet enters. It has four output ports where the packet is driven out.



### Packet Format:

Packet contains 3 parts. They are Header, data and frame check sequence.

Packet width is 8 bits and the length of the packet can be between 4 bytes to 259 bytes.

#### Packet header:

Packet header contains three fields DA, SA and length.

② DA: Destination address of the packet is of 8 bits. The switch drives the packet to respective ports based on this destination address of the packets. Each output port has 8-bit unique port address. If the destination address of the packet matches the port address, then switch drives the packet to the output port.

② SA: Source address of the packet from where it originate. It is 8 bits.

Length: Length of the data is of 8 bits and from 0 to 255. Length is measured in terms of bytes.

If Length = 0, it means data length is 0 bytes

If Length = 1, it means data length is 1 bytes

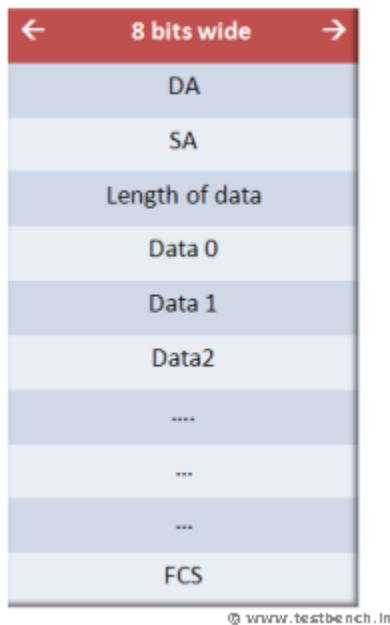
If Length = 2, it means data length is 2 bytes

If Length = 255, it means data length is 255 bytes

② Data: Data should be in terms of bytes and can take anything.

### FCS: Frame check sequence

This field contains the security check of the packet. It is calculated over the header and data.



© www.testbench.in

### Configuration:

Switch has four output ports. These output ports address have to be configured to a unique address. Switch matches the DA field of the packet with this configured port address and sends the packet on to that port. Switch contains a memory. This memory has 4 locations, each can store 8 bits. To configure the switch port address, memory write operation has to be done using memory interface. Memory address (0,1,2,3) contains the address of port(0,1,2,4) respectively.

### Interface Specification:

The Switch has one input Interface, from where the packet enters and 4 output interfaces from where the packet comes out and one memory interface, through the port address can be configured. Switch also has a clock and asynchronous reset signal.

### MEMORY INTERFACE:

Through memory interfaced output port address are configured. It accepts 8 bit data to be written to memory. It has 8 bit address inputs. Address 0,1,2,3 contains the address of the port 0,1,2,3 respectively.

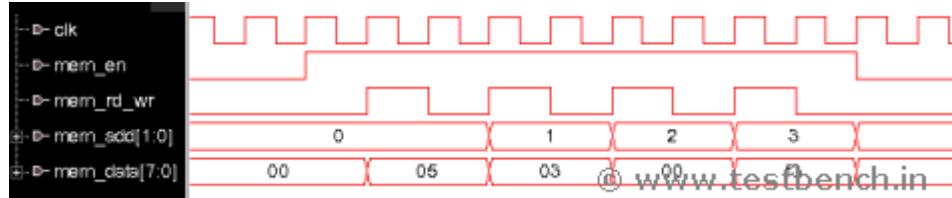
There are 4 input signals to memory interface. They are

```
input mem_en;  
input mem_rd_wr;  
input [1:0] mem_add;  
input [7:0] mem_data;
```

All the signals are active high and are synchronous to the positive edge of clock signal.

To configure a port address,

1. Assert the mem\_en signal.
2. Asser the mem\_rd\_wr signal.
3. Drive the port number (0 or 1 or 2 or 3) on the mem\_add signal
4. Drive the 8 bit port address on to mem\_data signal.



## INPUT PORT

Packets are sent into the switch using input port.

All the signals are active high and are synchronous to the positive edge of clock signal.

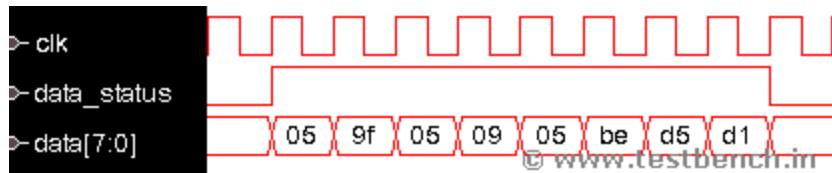
**input** port has **2 input** signals. They are

**input** [7:0] data;

**input** data\_status;

To send the packet in to switch,

1. Assert the data\_status signal.
2. Send the packet on the data signal byte by byte.
3. After sending all the data bytes, deassert the data\_status signal.
4. There should be at least 3 clock cycles difference between packets.



## OUTPUT PORT

Switch sends the packets out using the output ports. There are 4 ports, each having data, ready and read signals. All the signals are active high and are synchronous to the positive edge of clock signal.

Signal list is

**output** [7:0] port0;

**output** [7:0] port1;

**output** [7:0] port2;

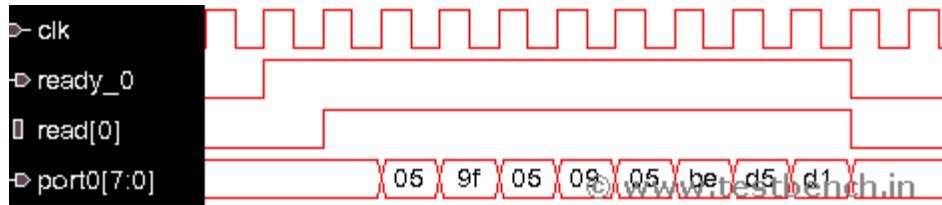
```

output [7:0] port3;
output ready_0;
output ready_1;
output ready_2;
output ready_3;
input read_0;
input read_1;
input read_2;
input read_3;

```

When the data is ready to be sent out from the port, switch asserts ready\_\* signal high indicating that data is ready to be sent.

If the read\_\* signal is asserted, when ready\_\* is high, then the data comes out of the port\_\* signal after one clock cycle.



### RTL code:

RTL code is attached with the tar files. From the Phase 1, you can download the tar files.

## VERIFICATION PLAN

### Overview

This Document describes the Verification Plan for Switch. The Verification Plan is based on System Verilog Hardware Verification Language. The methodology used for Verification is Constraint random coverage driven verification.

### Feature Extraction

This section contains list of all the features to be verified.

1)ID: Configuration

Description: Configure all the 4 port address with unique values.

2)ID: Packet DA

Description: DA field of packet should be any of the port address. All the 4 port address should be used.

3) ID : Packet payload

Description: Length can be from 0 to 255. Send packets with all the lengths.

4) ID: Length

Description:

Length field contains length of the payload.

Send Packet with correct length field and incorrect length fields.

## 5) ID: FCS

Description:

Good FCS: Send packet with good FCS.

Bad FCS: Send packet with corrupted FCS.

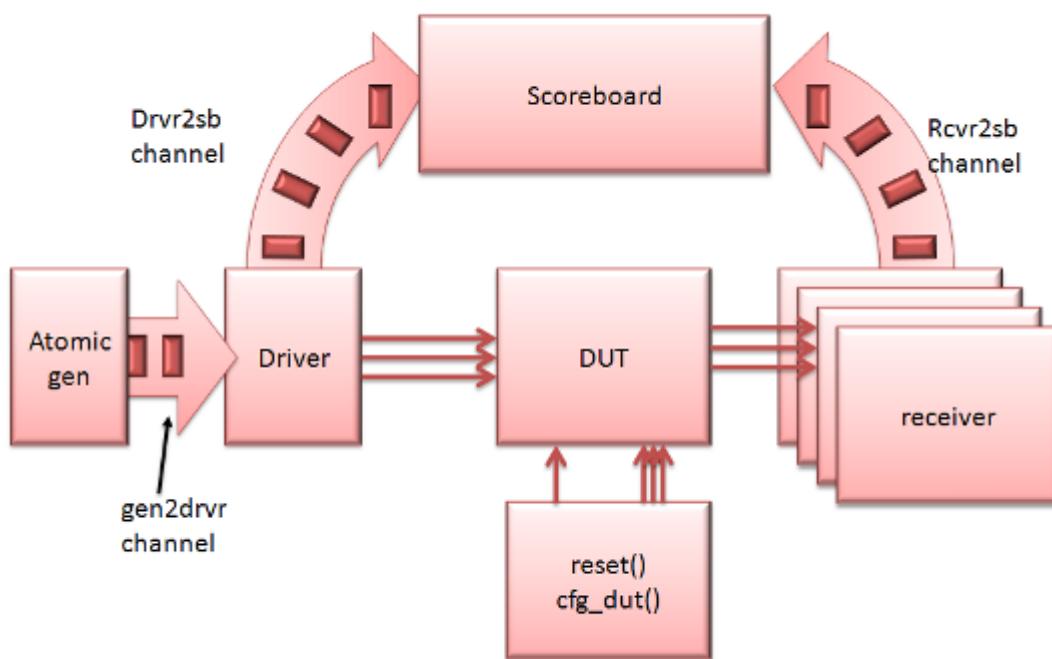
### Stimulus Generation Plan

- 1) Packet DA: Generate packet DA with the configured address.
- 2) Payload length: generate payload length ranging from 2 to 255.
- 3) Correct or Incorrect Length field.
- 4) Generate good and bad FCS.

### Coverage Plan

- 1) Cover all the port address configurations.
- 2) Cover all the packet lengths.
- 3) Cover all correct and incorrect length fields.
- 4) Cover good and bad FCS.
- 5) Cover all the above combinations.

### Verification Environment



## PHASE 1 TOP

In phase 1,

- 1) We will write SystemVerilog Interfaces for input port, output port and memory port.
- 2) We will write Top module where testcase and DUT instances are done.
- 3) DUT and TestBench interfaces are connected in top module.
- 4) Clock is generator in top module.

**NOTE:** In every file you will see the syntax

```
`ifndef GUARD_*
`endif GUARD_*
```

## Interfaces

In the interface.sv file, declare the 3 interfaces in the following way.

- ⌚ All the interfaces has clock as input.
- ⌚ All the signals in interface are wire type.
- ⌚ All the signals are synchronized to clock except reset in clocking block.

This approach will avoid race conditions between the design and the verification environment.

Define the set-up and hold time using parameters.

- ⌚ Signal directional w.r.t TestBench is specified with modport.

```
`ifndef GUARD_INTERFACE
`define GUARD_INTERFACE
///////////////////////////////
// Interface declaration for the memory///
///////////////////////////////

interface mem_interface(input bit clock);

parameter setup_time = 5ns;
parameter hold_time = 3ns;
wire [7:0] mem_data;
wire [1:0] mem_add;
wire mem_en;
wire mem_rd_wr;

clocking cb@(posedge clock);
  default input #setup_time output #hold_time;
  output mem_data;
  output mem_add;
  output mem_en;
  output mem_rd_wr;
endclocking:cb
```

```

modport MEM(clocking cb,input clock);

endinterface :mem_interface

///////////////////////////////
// Interface for the input side of switch.//
// Reset signal is also passed here.    //
///////////////////////////////

interface input_interface(input bit clock);

parameter setup_time = 5ns;
parameter hold_time = 3ns;

wire      data_status;
wire [7:0] data_in;
wire      reset;

clocking cb@(posedge clock);
  default input #setup_time output #hold_time;
  output   data_status;
  output   data_in;
endclocking:cb

modport IP(clocking cb,output reset,input clock);

endinterface:input_interface

///////////////////////////////
// Interface for the output side of the switch.//
// output_interface is for only one output port//
///////////////////////////////

interface output_interface(input bit clock);

parameter setup_time = 5ns;
parameter hold_time = 3ns;

wire [7:0] data_out;
wire ready;

```

```

wire read;
clocking cb@(posedge clock);
  default input #setup_time output #hold_time;
    input data_out;
    input ready;
    output read;
  endclocking:cb
  modport OP(clocking cb,input clock);
endinterface:output_interface
///////////
`endif

```

### Testcase

Testcase is a program block which provides an entry point for the test and creates a scope that encapsulates program-wide data. Currently this is an empty testcase which just ends the simulation. Program block contains all the above declared interfaces as arguments. This testcase has initial and final blocks.

Inside the program block, include the vmm.sv file.

```

`ifndef GUARD_TESTCASE
`define GUARD_TESTCASE
program testcase(mem_interface.MEM mem_intf,input_interface.IP input_intf,output_interface.
OP output_intf[4]);
`include "vmm.sv"
initial
begin
$display(" ***** Start of testcase *****");
#1000;
end

final
$display(" ***** End of testcase *****");

endprogram
`endif

```

### Top Module

The modules that are included in the source text but are not instantiated are called top modules. This module is the highest scope of modules. Generally this module is named as "top" and referenced as "top module". Module name can be anything.

This top-level module will contain the design portion of the simulation.

Do the following in the top module:

1)Generate the clock signal.

```
bit Clock;  
initial  
  begin  
    #20;  
    forever #10 Clock = ~Clock;  
  end
```

2)Do the instances of memory interface.

```
mem_interface mem_intf(Clock);
```

3)Do the instances of input interface.

```
input_interface input_intf(Clock);
```

4)There are 4 output ports. So do 4 instances of output\_interface.

```
output_interface output_intf[4](Clock);
```

5)Do the instance of testcase and pass all the above declared interfaces.

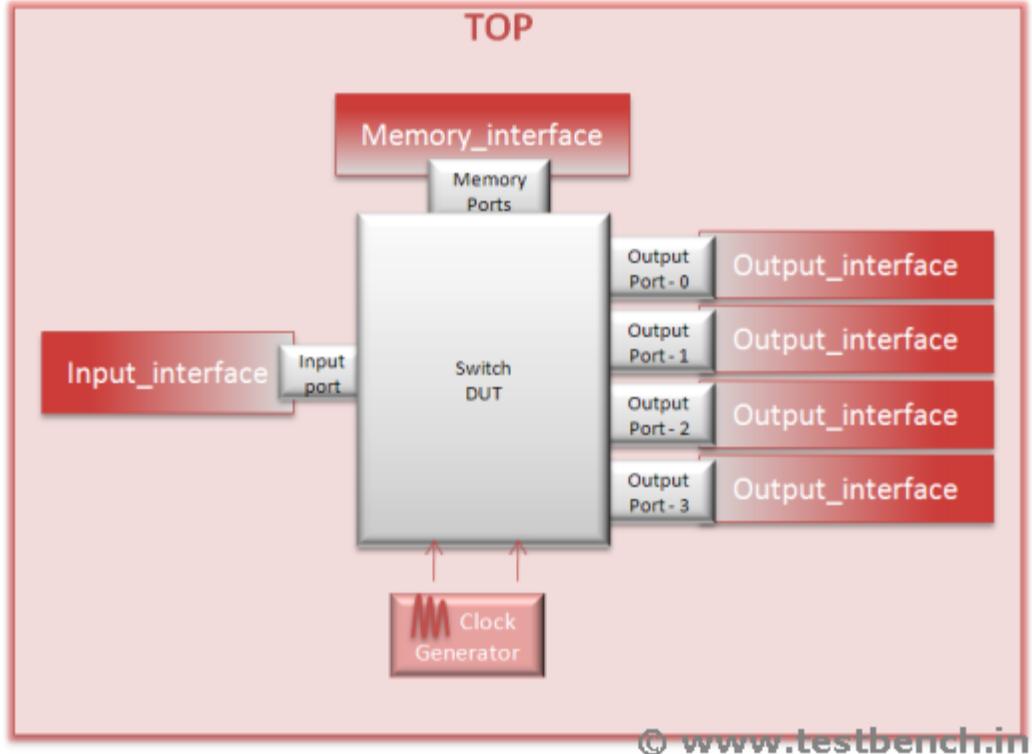
```
testcase TC (mem_intf,input_intf,output_intf);
```

6)Do the instance of DUT.

```
switch DUT (.
```

7)Connect all the interfaces and DUT. The design which we have taken is in verilog. So Verilog DUT instance is connected signal by signal.

```
switch DUT (.clk(Clock),  
  .reset(input_intf.reset),  
  .data_status(input_intf.data_status),  
  .data(input_intf.data_in),  
  .port0(output_intf[0].data_out),  
  .port1(output_intf[1].data_out),  
  .port2(output_intf[2].data_out),  
  .port3(output_intf[3].data_out),  
  .ready_0(output_intf[0].ready),  
  .ready_1(output_intf[1].ready),  
  .ready_2(output_intf[2].ready),  
  .ready_3(output_intf[3].ready),  
  .read_0(output_intf[0].read),  
  .read_1(output_intf[1].read),  
  .read_2(output_intf[2].read),  
  .read_3(output_intf[3].read),  
  .mem_en(mem_intf.mem_en),  
  .mem_rd_wr(mem_intf.mem_rd_wr),  
  .mem_add(mem_intf.mem_add),  
  .mem_data(mem_intf.mem_data));
```



### Top Module Source Code:

```

`ifndef GUARD_TOP
`define GUARD_TOP
module top();

///////////////////////////////
// Clock Declaration and Generation          //
///////////////////////////////
bit Clock;

initial
begin
#20;
forever #10 Clock = ~Clock;
end
///////////////////////////////
// Memory interface instance          //
/////////////////////////////

```

```

mem_interface mem_intf(Clock);
/////////////////////////////////////////////////////////////////
// Input interface instance          //
/////////////////////////////////////////////////////////////////

input_interface input_intf(Clock);
/////////////////////////////////////////////////////////////////
// output interface instance         //
/////////////////////////////////////////////////////////////////

output_interface output_intf[4](Clock);
/////////////////////////////////////////////////////////////////
// Program block Testcase instance  //
/////////////////////////////////////////////////////////////////

testcase TC (mem_intf,input_intf,output_intf);
/////////////////////////////////////////////////////////////////
// DUT instance and signal connection //
/////////////////////////////////////////////////////////////////


switch DUT  (.clk(Clock),
    .reset(input_intf.reset),
    .data_status(input_intf.data_status),
    .data(input_intf.data_in),
    .port0(output_intf[0].data_out),
    .port1(output_intf[1].data_out),
    .port2(output_intf[2].data_out),
    .port3(output_intf[3].data_out),
    .ready_0(output_intf[0].ready),
    .ready_1(output_intf[1].ready),
    .ready_2(output_intf[2].ready),
    .ready_3(output_intf[3].ready),
    .read_0(output_intf[0].read),
    .read_1(output_intf[1].read),
    .read_2(output_intf[2].read),
    .read_3(output_intf[3].read),

```

```
.mem_en(mem_intf.mem_en),  
.mem_rd_wr(mem_intf.mem_rd_wr),  
.mem_add(mem_intf.mem_add),  
.mem_data(mem_intf.mem_data));
```

**endmodule**

`endif

Download the phase 1 files:

[vmm\\_switch\\_1.tar](#)

[Browse the code in vmm\\_switch\\_1.tar](#)

Run the simulation:

vcs -sverilog -f filelist -R -ntb\_opts rvm

Log file after simulation:

```
***** Start of testcase *****  
***** End of testcase *****
```

## **PHASE 2 ENVIRONMENT**

In this phase, we will write

- ⌚ Environment class.
- ⌚ Virtual interface declaration.
- ⌚ Defining Environment class constructor.
- ⌚ Defining required methods for execution . Currently these methods will not be implemented in this phase.

All the above are done in Environment .sv file.

We will write a testcase using the above define environment class in testcase.sv file.

### **Environment Class:**

The class is a base class used to implement verification environments.

Environment class is extension on vmm\_env class. The testbench simulation needs some systematic flow like reset, initialize etc. vmm\_env base class has methods formalize the simulation steps. All methods are declared as virtual methods.

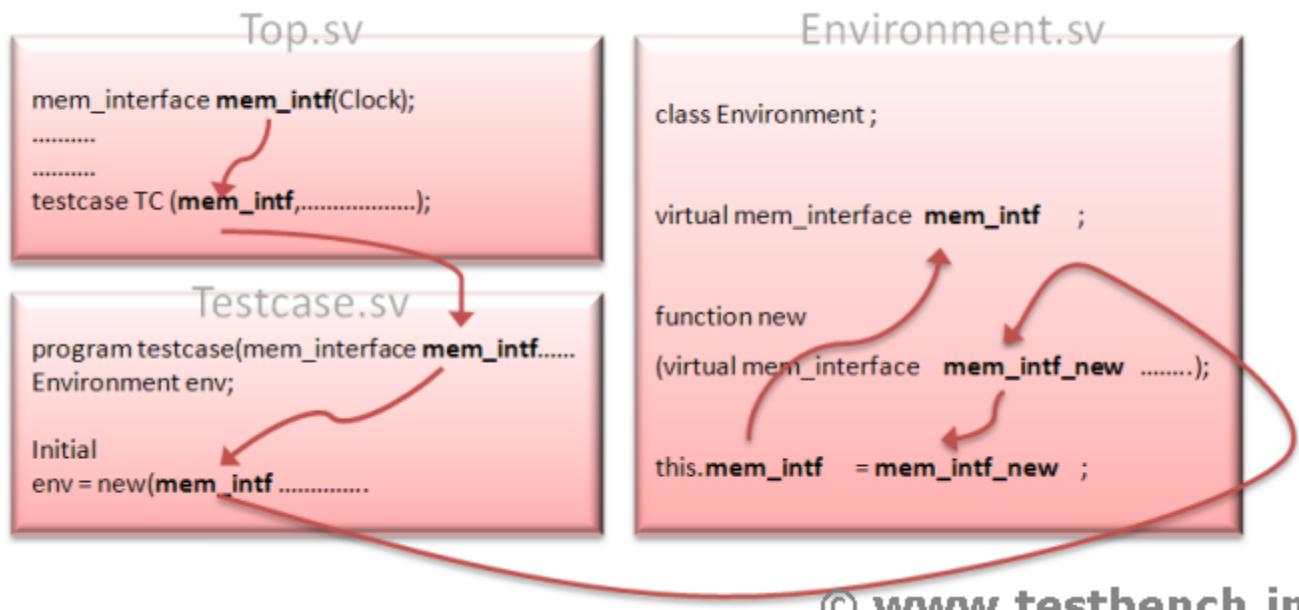
We will not implement all the vmm\_env virtual methods in this phase but will we print messages from these methods to understand the simulation execution.

Testcase contains the instance of the environment class and has access to all the public declaration of environment class. This testcase Creates a Environment object and calls the run() method which are defined in the environment class. Run() method runs all the simulation methods which are defined in environment class.

Verification environment contains the declarations of the virtual interfaces. These virtual interfaces are pointed to the physical interfaces which are declared in the top module.

Constructor method should be declared with virtual interface as arguments, so that when the object is created, in the testcase can pass the interfaces in to environment class.

Connecting the virtual interfaces of Environment class to the physical interfaces of top module.



© www.testbench.in

Extend vmm\_env class to define Environment class.

```

`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends vmm_env;

```

```
endclass
```

```
`endif
```

Declare virtual interfaces in Environment class.

```
virtual mem_interface.MEM mem_intf ;  
virtual input_interface.IP input_intf ;  
virtual output_interface.OP output_intf[4] ;
```

The construction of Environment class is declared with virtual interface as arguments.

```
function new(virtual mem_interface.MEM mem_intf_new ,  
           virtual input_interface.IP input_intf_new ,  
           virtual output_interface.OP output_intf_new[4]);
```

```
super.new("Environment");
```

In constructor methods, the interfaces which are arguments are connected to the virtual interfaces of environment class.

```
this.mem_intf = mem_intf_new ;  
this.input_intf = input_intf_new ;  
this.output_intf = output_intf_new ;
```

```
`vmm_note(this.log, "Created env object");
```

Extend all the vmm\_env virtual methods. Call the super.<method\_name> in all the vmm\_env virtual method extensions.

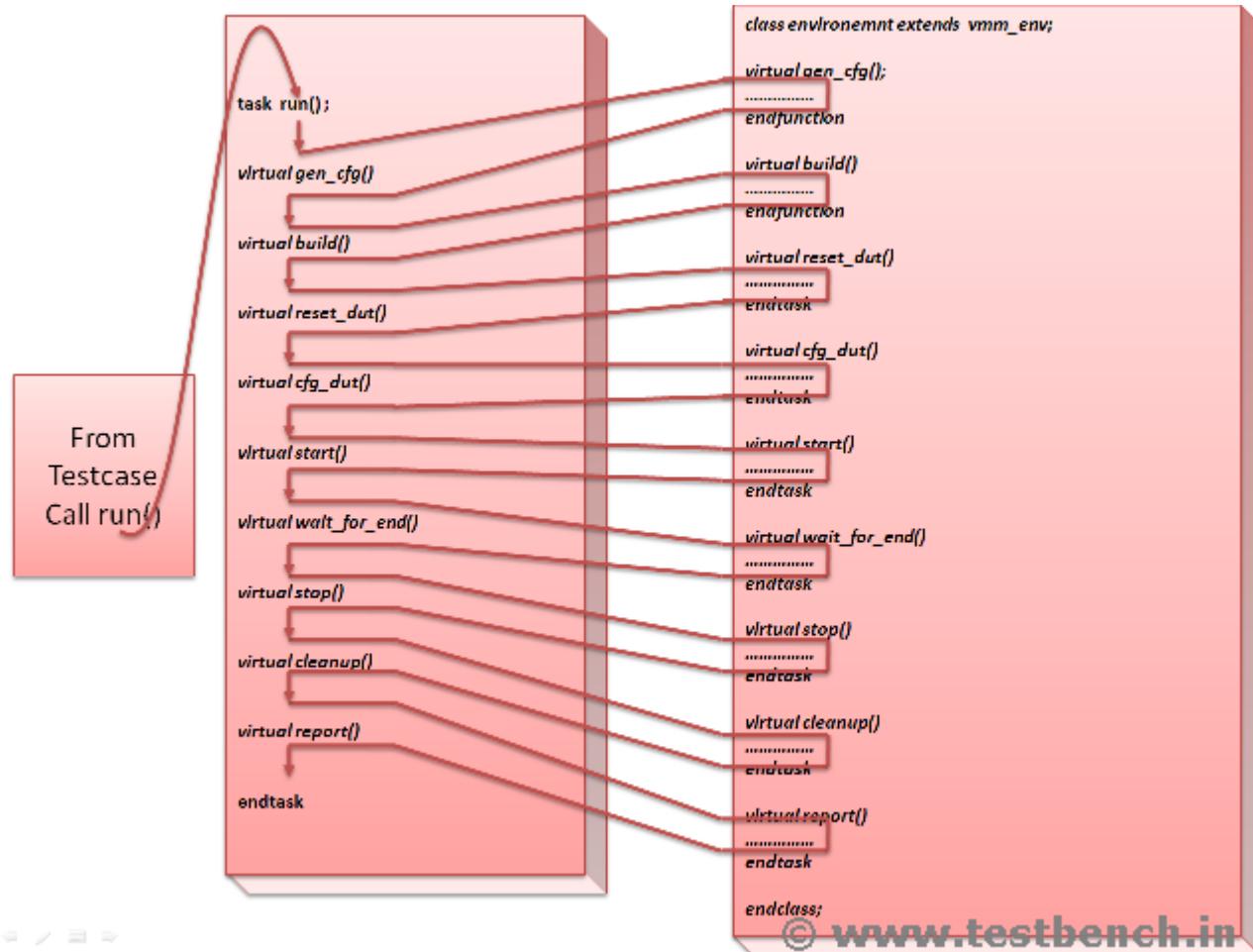
Include `vmm\_note() messages for identifying the simulation steps in the log file.

```
virtual function void gen_cfg();  
  super.gen_cfg();  
  `vmm_note(this.log,"Start of gen_cfg() method ");  
  `vmm_note(this.log,"End of gen_cfg() method ");  
endfunction  
virtual function void build();  
  super.build();  
  `vmm_note(this.log,"Start of build() method ");  
  `vmm_note(this.log,"End of build() method ");  
endfunction  
....  
....  
....
```

Do the above for reset\_dut(), cfg\_dut(), start(), wait\_for\_end(), stop() cleanup() and report() methods.

### Run :

The run method is called from the testcase to start the simulation. run method calls all the methods which are defined in the Environment class.



### Environment Class Source Code:

```
`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends vmm_env;
    virtual mem_interface.MEM mem_intf ;
    virtual input_interface.IP input_intf ;
    virtual output_interface.OP output_intf[4] ;
```

```

function new(virtual mem_interface.MEM mem_intf_new ,
    virtual input_interface.IP input_intf_new ,
    virtual output_interface.OP output_intf_new[4] );
super.new("Environment");
this.mem_intf = mem_intf_new ;
this.input_intf = input_intf_new ;
this.output_intf = output_intf_new ;

`vmm_note(this.log, "Created env object");
endfunction : new

virtual function void gen_cfg();
    super.gen_cfg();
    `vmm_note(this.log,"Start of gen_cfg() method ");
    `vmm_note(this.log,"End of gen_cfg() method ");
endfunction

virtual function void build();
    super.build();
    `vmm_note(this.log,"Start of build() method ");
    `vmm_note(this.log,"End of build() method ");
endfunction

virtual task reset_dut();
    super.reset_dut();
    `vmm_note(this.log,"Start of reset_dut() method ");
    `vmm_note(this.log,"End of reset_dut() method ");
endtask

virtual task cfg_dut();
    super.cfg_dut();
    `vmm_note(this.log,"Start of cfg_dut() method ");
    `vmm_note(this.log,"End of cfg_dut() method ");
endtask

virtual task start();
    super.start();
    `vmm_note(this.log,"Start of start() method ");
    `vmm_note(this.log,"End of start() method ");
endtask

virtual task wait_for_end();

```

```

super.wait_for_end();
`vmm_note(this.log,"Start of wait_for_end() method ");
`vmm_note(this.log,"End of wait_for_end() method ");
endtask

virtual task stop();
super.stop();
`vmm_note(this.log,"Start of stop() method ");
`vmm_note(this.log,"End of stop() method ");
endtask

virtual task cleanup();
super.cleanup();
`vmm_note(this.log,"Start of cleanup() method ");
`vmm_note(this.log,"End of cleanup() method ");
endtask

virtual task report();
`vmm_note(this.log,"Start of report() method \n\n");
$display("-----");
super.report();
$display("-----");
$display("\n\n");
`vmm_note(this.log,"End of report() method");
endtask

endclass
`endif

```

We will create a file Global.sv for global requirement. In this file, define all the port address as macros in this file.

```

`ifndef GUARD_GLOBALS
`define GUARD_GLOBALS

`define P0 8'h00
`define P1 8'h11

```

```
`define P2 8'h22  
`define P3 8'h33
```

```
`endif
```

Now We will update the testcase. Take an instance of the Environment class and call the run method of the Environment class.

```
`ifndef GUARD_TESTCASE  
`define GUARD_TESTCASE
```

```
program testcase(mem_interface.MEM mem_intf,input_interface.IP  
input_intf,output_interface.OP output_intf[4]);  
`include "vmm.sv"  
Environment env;  
initial  
begin  
$display(" ***** Start of testcase *****");  
  
env = new(mem_intf,input_intf,output_intf);  
env.run();  
end  
final  
$display(" ***** End of testcase *****");  
endprogram  
`endif
```

#### Download the phase 2 source code:

[vmm\\_switch\\_2.tar](#)

[Browse the code in vmm\\_switch\\_2.tar](#)

#### Run the simulation:

vcs -sverilog -f filelist -R -ntb\_opts rvm

#### Log report after the simulation:

```
***** Start of testcase *****  
Normal[NOTE] on Environment() at 0:  
    Created env object  
Normal[NOTE] on Environment() at 0:  
    Start of gen_cfg() method  
Normal[NOTE] on Environment() at 0:  
    End of gen_cfg() method  
Normal[NOTE] on Environment() at 0:
```

Start of build() method	
Normal[NOTE] on Environment() at	0:
End of build() method	
Normal[NOTE] on Environment() at	0:
Start of reset_dut() method	
Normal[NOTE] on Environment() at	0:
End of reset_dut() method	
Normal[NOTE] on Environment() at	0:
Start of cfg_dut() method	
Normal[NOTE] on Environment() at	0:
End of cfg_dut() method	
Normal[NOTE] on Environment() at	0:
Start of start() method	
Normal[NOTE] on Environment() at	0:
End of start() method	
Normal[NOTE] on Environment() at	0:
Start of wait_for_end() method	
Normal[NOTE] on Environment() at	0:
End of wait_for_end() method	
Normal[NOTE] on Environment() at	0:
Start of stop() method	
Normal[NOTE] on Environment() at	0:
End of stop() method	
Normal[NOTE] on Environment() at	0:
Start of cleanup() method	
Normal[NOTE] on Environment() at	0:
End of cleanup() method	
Normal[NOTE] on Environment() at	0:
Start of report() method	

---

Simulation PASSED on ./ (./) at 0 (0 warnings, 0 demoted errors & 0 demoted warnings)

---

Normal[NOTE] on Environment() at 0:

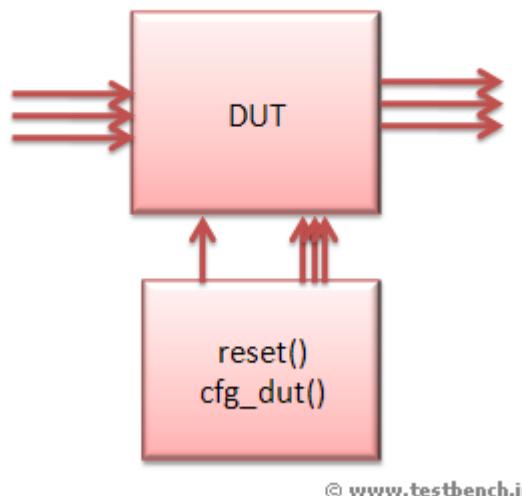
End of report() method

\*\*\*\*\* End of testcase \*\*\*\*\*

### **PHASE 3 RESET**

In this phase we will reset and configure the DUT.

The Environment class has reset\_dut() method which contains the logic to reset the DUT and cfg\_dut() method which contains the logic to configure the DUT port address.



**NOTE:** Clocking block signals can be driven only using a non-blocking assignment.

In reset\_dut() method.

- 1) Set all the DUT input signals to a known state. And reset the DUT.

```
virtual task reset_dut();
    super.reset_dut();
    `vvm_note(this.log,"Start of reset_dut() method ");

    mem_intf.cb.mem_data    <= 0;
    mem_intf.cb.mem_add     <= 0;
    mem_intf.cb.mem_en      <= 0;
    mem_intf.cb.mem_rd_wr   <= 0;
    input_intf.cb.data_in   <= 0;
    input_intf.cb.data_status <= 0;
    output_intf[0].cb.read   <= 0;
```

```

output_intf[1].cb.read  <= 0;
output_intf[2].cb.read  <= 0;
output_intf[3].cb.read  <= 0;

// Reset the DUT
input_intf.reset      <= 1;
repeat (4) @ input_intf.clock;
input_intf.reset      <= 0;

`vmm_note(this.log,"End of reset_dut() method ");
endtask

```

2) Updated the cfg\_dut method.

```

virtual task cfg_dut();
    super.cfg_dut();
    `vmm_note(this.log,"Start of cfg_dut() method ");

    mem_intf.cb.mem_en <= 1;
    @(posedge mem_intf.clock);
    mem_intf.cb.mem_rd_wr <= 1;

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h0;
    mem_intf.cb.mem_data <= `P0;
    `vmm_note(this.log ,\$psprintf(" Port 0 Address %h ",`P0));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h1;
    mem_intf.cb.mem_data <= `P1;
    `vmm_note(this.log ,\$psprintf(" Port 1 Address %h ",`P1));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h2;
    mem_intf.cb.mem_data <= `P2;
    `vmm_note(this.log ,\$psprintf(" Port 2 Address %h ",`P2));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h3;
    mem_intf.cb.mem_data <= `P3;
    `vmm_note(this.log ,\$psprintf(" Port 3 Address %h ",`P3));

```

```

@(posedge mem_intf.clock);
mem_intf.cb.mem_en  <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add  <= 0;
mem_intf.cb.mem_data <= 0;

`vmm_note(this.log,"End of cfg_dut() method ");
endtask

```

(3) In wait\_for\_end method, wait for some clock cycles.

```
repeat(10000) @(input_intf.clock);
```

#### Download the Phase 3 source code:

[vmm\\_switch\\_3.tar](#)

[Browse the code in vmm\\_switch\\_3.tar](#)

#### Run the simulation:

```
vcs -sverilog -f filelist -R -ntb_opts rvm
```

#### Log File report

```
***** Start of testcase *****
Normal[NOTE] on Environment() at      0:
    Created env object
Normal[NOTE] on Environment() at      0:
    Start of gen_cfg() method
Normal[NOTE] on Environment() at      0:
    End of gen_cfg() method
Normal[NOTE] on Environment() at      0:
    Start of build() method
Normal[NOTE] on Environment() at      0:
    End of build() method
Normal[NOTE] on Environment() at      0:
    Start of reset_dut() method
Normal[NOTE] on Environment() at      60:
    End of reset_dut() method
```

Normal[NOTE] on Environment() at	60:
Start of cfg_dut() method	
Normal[NOTE] on Environment() at	90:
Port 0 Address 00	
Normal[NOTE] on Environment() at	110:
Port 1 Address 11	
Normal[NOTE] on Environment() at	130:
Port 2 Address 22	
Normal[NOTE] on Environment() at	150:
Port 3 Address 33	
Normal[NOTE] on Environment() at	170:
End of cfg_dut() method	
Normal[NOTE] on Environment() at	170:
Start of start() method	
Normal[NOTE] on Environment() at	170:
End of start() method	
Normal[NOTE] on Environment() at	170:
Start of wait_for_end() method	
Normal[NOTE] on Environment() at	100170:
End of wait_for_end() method	
Normal[NOTE] on Environment() at	100170:
Start of stop() method	
Normal[NOTE] on Environment() at	100170:
End of stop() method	
Normal[NOTE] on Environment() at	100170:
Start of cleanup() method	
Normal[NOTE] on Environment() at	100170:
End of cleanup() method	
Normal[NOTE] on Environment() at	100170:
Start of report() method	

---

Simulation PASSED on ./ (./) at 100170 (0 warnings, 0 demoted errors & 0 demoted warnings)

---

Normal[NOTE] on Environment() at 100170:  
End of report() method  
\*\*\*\*\* End of testcase \*\*\*\*\*

## **PHASE 4 PACKET**

In this Phase , We will define a packet and then test it whether it is generating as expected.

Packet is modeled using class. Packet class should be able to generate all possible packet types randomly. Packet class should also implement allocate(), psdisplay(), copy(), compare(), byte\_pack() and byte\_unpack() of vmm\_data methods.

Using vmm macros, we will create atomic generator and channel for Packet.

We will write the packet class in Packet.sv file. Packet class variables and constraints have been derived from stimulus generation plan.

Revisit Stimulus Generation Plan

- 1) Packet DA: Generate packet DA with the configured address.
- 2) Payload length: generate payload length ranging from 2 to 255.
- 3) Correct or Incorrect Length field.
- 4) Generate good and bad FCS.

1) Declare FCS types as enumerated data types. Name members as GOOD\_FCS and BAD\_FCS.

**typedef enum { GOOD\_FCS, BAD\_FCS } fcs\_kind\_t;**

2) Declare the length type as enumerated data type. Name members as GOOD\_LENGTH and BAD\_LENGTH.

**typedef enum { GOOD\_LENGTH, BAD\_LENGTH } length\_kind\_t;**

3) Extend vmm\_data class to define Paclet class.

**class Packet extends vmm\_data;**

4) Declare a vmm\_log object and construct it.

**static vmm\_log log = new("Packet","Class");**

5) Declare the length type and fcs type variables as rand.

**rand fcs\_kind\_t fcs\_kind;**

**rand length\_kind\_t length\_kind;**

6) Declare the packet field as rand. All fields are bit data types. All fields are 8 bit packet array.

Declare the payload as dynamic array.

**rand bit [7:0] length;**

**rand bit [7:0] da;**

**rand bit [7:0] sa;**

**rand bit [7:0] data[]; //Payload using Dynamic array,size is generated on the fly**

**rand byte fcs;**

7) Constraint the DA field to be any one of the configured address.

**constraint address\_c { da inside { `P0,`P1,`P2,`P3 } ; }**

8) Constrain the payload dynamic array size to between 1 to 255.

**constraint payload\_size\_c { data.size inside { [1 : 255] }; }**

9) Constrain the payload length to the length field based on the length type.

```
constraint length_kind_c {
    (length_kind == GOOD_LENGTH) -> length == data.size;
    (length_kind == BAD_LENGTH) -> length == data.size + 2 ; }
```

Use solve before to direct the randomization to generate first the payload dynamic array size and then randomize length field.

```
constraint solve_size_length { solve data.size before length; }
```

10) Define a port\_randomize method. In this method calculate the fcs based on the fcs\_kind.

```
function void post_randomize();
    if(fcs_kind == GOOD_FCS)
        fcs = 8'b0;
    else
        fcs = 8'b1;
    fcs = cal_fcs();
endfunction : post_randomize
```

11) Define the FCS method.

```
virtual function byte cal_fcs;
    integer i;
    byte result ;
    result = 0;
    result = result ^ da;
    result = result ^ sa;
    result = result ^ length;
    for (i=0;i< data.size;i++)
        result = result ^ data[i];
    result = fcs ^ result;
    return result;
endfunction : cal_fcs
```

12) Define acallocate() method.

```
virtual function vmm_data allocate();
    Packet pkt;
    pkt = new();
    return pkt;
endfunction:allocate
```

13) Define psdisplay() method. psdisplay() method displays the current value of the packet fields to a string.

```
virtual function string psdisplay(string prefix = "");
    int i;
```

```

$write(psdisplay, " %s packet
#%0d.%0d.%0d\n", prefix,this.stream_id, this.scenario_id, this.data_id);
$write(psdisplay, " %s%s da:0x%h\n", psdisplay, prefix,this.da);
$write(psdisplay, " %s%s sa:0x%h\n", psdisplay, prefix,this.sa);
$write(psdisplay, " %s%s length:0x%h
(data.size=%0d)\n", psdisplay, prefix,this.length,this.data.size());
$write(psdisplay, " %s%s data[%0d]:0x%h", psdisplay, prefix,0,data[0]);
if(data.size() > 1)
    $write(psdisplay, " data[%0d]:0x%h", 1,data[1]);
if(data.size() > 4)
    $write(psdisplay, " .... ");
if(data.size() > 2)
    $write(psdisplay, " data[%0d]:0x%h", data.size() -2,data[data.size() -2]);
if(data.size() > 3)
    $write(psdisplay, " data[%0d]:0x%h", data.size() -1,data[data.size() -1]);
$write(psdisplay, "\n %s%s fcs:0x%h \n", psdisplay, prefix, this.fcs);

```

### **endfunction**

14) Define copy() method. copy() method copies the current values of the object instance.

```

virtual function vmm_data copy(vmm_data to = null);
    Packet cpy;

    // Copying to a new instance?
    if (to == null)
        cpy = new;
    else

        // Copying to an existing instance. Correct type?
        if (!$cast(cpy, to))
            begin
                `vmm_fatal(this.log, "Attempting to copy to a non packet instance");
                copy = null;
            return copy;
        end

    super.copy_data(cpy);

```

```

cpy.da = this.da;
cpy.sa = this.sa;
cpy.length = this.length;
cpy.data = new[this.data.size()];
foreach(data[i])
    begin
        cpy.data[i] = this.data[i];
    end
    cpy.fcs = this.fcs;
    copy = cpy;
endfunction:copy

```

15) Define Compare() method. Compares the current value of the object instance with the specified object instance.

If the value is different, FALSE is returned.

```

virtual function bit compare(input vmm_data to,output string diff,input int kind= -1);
Packet cmp;

compare = 1; // Assume success by default.
diff = "No differences found";

if (!$cast(cmp, to))
begin
    `vmm_fatal(this.log, "Attempting to compare to a non packet instance");
    compare = 0;
    diff = "Cannot compare non packets";
    return compare;
end

// data types are the same, do comparison:
if (this.da != cmp.da)
begin
    diff = $psprintf("Different DA values: %b != %b", this.da, cmp.da);
    compare = 0;
    return compare;
end

if (this.sa != cmp.sa)

```

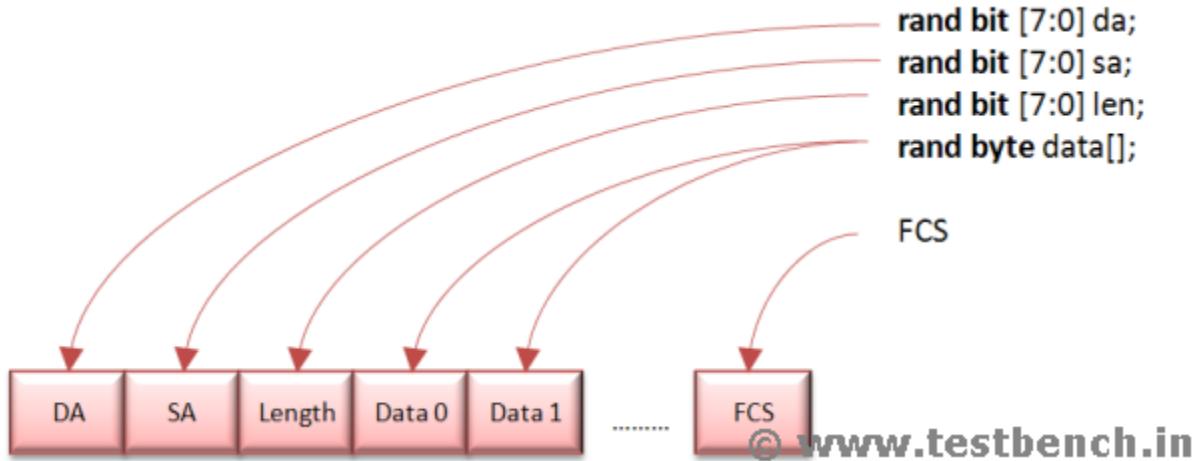
```

begin
    diff = $psprintf("Different SA values: %b != %b", this.sa, cmp.sa);
    compare = 0;
    return compare;
end
if (this.length != cmp.length)
begin
    diff = $psprintf("Different LEN values: %b != %b", this.length, cmp.length);
    compare = 0;
    return compare;
end

foreach(data[i])
    if (this.data[i] != cmp.data[i])
        begin
            diff = $psprintf("Different data[%0d] values: 0x%h != 0x%h", i, this.data[i], cmp.data[i]);
            compare = 0;
            return compare;
        end
    if (this.fcs != cmp.fcs)
        begin
            diff = $psprintf("Different FCS values: %b != %b", this.fcs, cmp.fcs);
            compare = 0;
            return compare;
        end
endfunction:compare

```

16)Define byte\_pack() method().



Packing is commonly used to convert the high level data to low level data that can be applied to DUT. In packet class various fields are generated. Required fields are concatenated to form a stream of bytes which can be driven conveniently to DUT interface by the driver.

```

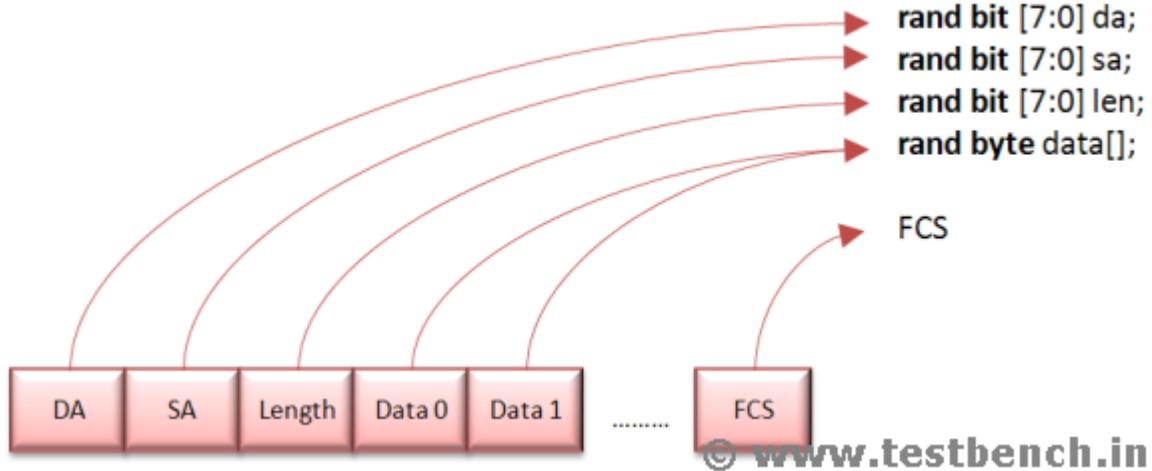
virtual function int unsigned byte_pack(
    ref logic [7:0] bytes[],
    input int unsigned offset =0 ,
    input int kind = -1);
    byte_pack = 0;
    bytes = new[this.data.size() + 4];
    bytes[0] = this.da;
    bytes[1] = this.sa;
    bytes[2] = this.length;

    foreach(data[i])
        bytes[3+i] = data[i];

    bytes[this.data.size() + 3 ] = fcs;
    byte_pack = this.data.size() + 4;
endfunction:byte_pack

```

17) Define byte\_unpack() method:



The unpack() method does exactly the opposite of pack method. Unpacking is commonly used to convert a data stream coming from DUT to high level data packet object.

```

virtual function int unsigned byte_unpack(
    const ref logic [7:0] bytes[],
    input int unsigned offset = 0,
    input int len = -1,
    input int kind = -1);
    this.da = bytes[0];
    this.sa = bytes[1];
    this.length = bytes[2];
    this.fcs = bytes[bytes.size() -1];
    this.data = new[bytes.size() - 4];
    foreach(data[i])
        this.data[i] = bytes[i+3];
    return bytes.size();
endfunction:byte_unpack

```

18) Define Packet\_channel for Packet using macro.

```
`vmm_channel(Packet)
```

19) Define Packet\_atomic\_gen for generating Packet instances using macro.

```
`vmm_atomic_gen(Packet, "Packet Gen")
```

## Packet Class Source Code

```
`ifndef GUARD_PACKET
`define GUARD_PACKET

//Define the enumerated types for packet types
typedef enum { GOOD_FCS, BAD_FCS } fcs_kind_t;
typedef enum { GOOD_LENGTH, BAD_LENGTH } length_kind_t;

class Packet extends vmm_data;

static vmm_log log = new("Packet","Class");

rand fcs_kind_t   fcs_kind;
rand length_kind_t length_kind;

rand bit [7:0] length;
rand bit [7:0] da;
rand bit [7:0] sa;
rand bit [7:0] data[];//Payload using Dynamic array,size is generated on the fly
rand byte fcs;

constraint address_c { da inside {'P0,'P1,'P2,'P3} ; }

constraint payload_size_c { data.size inside { [1 : 255];}

constraint length_kind_c {
    (length_kind == GOOD_LENGTH) -> length == data.size;
    (length_kind == BAD_LENGTH) -> length == data.size + 2 ; }

constraint solve_size_length { solve data.size before length; }

function new();
    super.new(this.log);
endfunction:new

function void post_randomize();
    if(fcs_kind == GOOD_FCS)
        fcs = 8'b0;
```

```

else
    fcs = 8'b1;
    fcs = cal_fcs();
endfunction : post_randomize

//// method to calculate the fcs /////
virtual function byte cal_fcs;
    integer i;
    byte result ;
    result = 0;
    result = result ^ da;
    result = result ^ sa;
    result = result ^ length;
    for (i = 0;i< data.size;i++)
        result = result ^ data[i];
    result = fcs ^ result;
    return result;
endfunction : cal_fcs

virtual function vmm_data allocate();
    Packet pkt;
    pkt = new();
    return pkt;
endfunction:allocate

virtual function string psdisplay(string prefix = "");
    int i;

    $write(psdisplay, " %s packet
#%0d.%0d.%0d\n", prefix,this.stream_id, this.scenario_id, this.data_id);
    $write(psdisplay, " %s%s da:0x%h\n", psdisplay, prefix,this.da);
    $write(psdisplay, " %s%s sa:0x%h\n", psdisplay, prefix,this.sa);
    $write(psdisplay, " %s%s length:0x%h
(data.size=%0d)\n", psdisplay, prefix,this.length,this.data.size());
    $write(psdisplay, " %s%s data[%0d]:0x%h", psdisplay, prefix,0,data[0]);
    if(data.size() > 1)
        $write(psdisplay, " data[%0d]:0x%h", 1,data[1]);
    if(data.size() > 4)
        $write(psdisplay, " .... ");
    if(data.size() > 2)

```

```

    $write(psdisplay, "  data[%0d]:0x%h", data.size() -2,data[data.size() -2]);
  if(data.size() > 3)
    $write(psdisplay, "  data[%0d]:0x%h", data.size() -1,data[data.size() -1]);
$write(psdisplay, "\n  %s%s  fcs:0x%h \n", psdisplay, prefix, this.fcs);

endfunction

```

```

virtual function vmm_data copy(vmm_data to = null);
Packet cpy;

// Copying to a new instance?
if (to == null)
  cpy = new;
else

// Copying to an existing instance. Correct type?
if (!$cast(cpy, to))
  begin
    `vmm_fatal(this.log, "Attempting to copy to a non packet instance");
    copy = null;
    return copy;
  end
  super.copy_data(cpy);
  cpy.da = this.da;
  cpy.sa = this.sa;
  cpy.length = this.length;
  cpy.data = new[this.data.size()];
  foreach(data[i])
    begin
      cpy.data[i] = this.data[i];
    end
    cpy.fcs = this.fcs;
    copy = cpy;
  endfunction:copy

```

```

virtual function bit compare(input vmm_data to,output string diff,input int kind= -1);
Packet cmp;

```

compare = 1; // Assume success by default.

```

diff = "No differences found";

if (!$cast(cmp, to))
begin
    `vmm_fatal(this.log, "Attempting to compare to a non packet instance");
    compare = 0;
    diff = "Cannot compare non packets";
    return compare;
end

// data types are the same, do comparison:
if (this.da != cmp.da)
begin
    diff = $psprintf("Different DA values: %b != %b", this.da, cmp.da);
    compare = 0;
    return compare;
end

if (this.sa != cmp.sa)
begin
    diff = $psprintf("Different SA values: %b != %b", this.sa, cmp.sa);
    compare = 0;
    return compare;
end

if (this.length != cmp.length)
begin
    diff = $psprintf("Different LEN values: %b != %b", this.length, cmp.length);
    compare = 0;
    return compare;
end

foreach(data[i])
    if (this.data[i] != cmp.data[i])
        begin
            diff = $psprintf("Different data[%0d] values: 0x%h != 0x%h", i, this.data[i], cmp.data[i]);
            compare = 0;
            return compare;
        end
    if (this.fcs != cmp.fcs)
        begin

```

```

diff = $psprintf("Different FCS values: %b != %b", this.fcs, cmp.fcs);
compare = 0;
return compare;
end
endfunction:compare

virtual function int unsigned byte_pack(
    ref logic [7:0] bytes[],
    input int unsigned offset = 0 ,
    input int kind = -1);
byte_pack = 0;
bytes = new[this.data.size() + 4];
bytes[0] = this.da;
bytes[1] = this.sa;
bytes[2] = this.length;

foreach(data[i])
    bytes[3+i] = data[i];

bytes[this.data.size() + 3 ] = fcs;
byte_pack = this.data.size() + 4;
endfunction:byte_pack

virtual function int unsigned byte_unpack(
    const ref logic [7:0] bytes[],
    input int unsigned offset = 0,
    input int len = -1,
    input int kind = -1);
this.da = bytes[0];
this.sa = bytes[1];
this.length = bytes[2];
this.fcs = bytes[bytes.size() -1];
this.data = new[bytes.size() - 4];
foreach(data[i])
    this.data[i] = bytes[i+3];
return bytes.size();
endfunction:byte_unpack

endclass

```

```
//////////  
/// Create vmm_channel and vmm_atomic_gen for packet///  
//////////  
`vmm_channel(Packet)  
`vmm_atomic_gen(Packet, "Packet Gen")
```

Now we will write a small program to test our packet implantation. This program block is not used to verify the DUT.

Write a simple program block and do the instance of packet class. Randomize the packet and call the display method to analyze the generation. Then pack the packet in to bytes and then unpack bytes and then call compare method to check all the methods.

### Program Block Source Code

```
program test;  
packet pkt1 = new();  
packet pkt2 = new();  
logic [7:0] bytes[];  
initial  
repeat(10)  
if(pkt1.randomize)  
begin  
$display(" Randomization Successes full.");  
pkt1.display();  
void'(pkt1.byte_pack(bytes));  
pkt2 = new();  
pkt2.byte_unpack(bytes);  
if(pkt2.compare(pkt1))  
$display(" Packing, Unpacking and compare worked");  
else  
$display(" *** Something went wrong in Packing or Unpacking or compare ***");  
end  
else  
$display(" *** Randomization Failed ***");  
endprogram
```

Download the packet class with program block.

vmm\_switch\_4.tar

Browse the code in vmm\_switch\_4.tar

Run the simulation

vcs -sverilog -f filelist -R -ntb\_opts rvm

### Log file report:

Randomization Sucessesfull.

```
Pkt1 packet #0.0.0
Pkt1 da:0x00
Pkt1 sa:0x40
Pkt1 length:0xbe (data.size=190)
Pkt1 data[0]:0xf7 data[1]:0xa6 .... data[188]:0x49 data[189]:0x79
Pkt1 fcs:0x1a
```

```
Pkt2 packet #0.0.0
Pkt2 da:0x00
Pkt2 sa:0x40
Pkt2 length:0xbe (data.size=190)
Pkt2 data[0]:0xf7 data[1]:0xa6 .... data[188]:0x49 data[189]:0x79
Pkt2 fcs:0x1a
```

Packing,Unpacking and compare worked

Randomization Sucessesfull.

```
Pkt1 packet #0.0.0
Pkt1 da:0x33
Pkt1 sa:0xfd
Pkt1 length:0xc7 (data.size=199)
Pkt1 data[0]:0x1e data[1]:0x80 .... data[197]:0x15 data[198]:0x30
Pkt1 fcs:0xa5
```

```
Pkt2 packet #0.0.0
Pkt2 da:0x33
Pkt2 sa:0xfd
Pkt2 length:0xc7 (data.size=199)
Pkt2 data[0]:0x1e data[1]:0x80 .... data[197]:0x15 data[198]:0x30
Pkt2 fcs:0xa5
```

Packing,Unpacking and compare worked

Randomization Sucessesfull.

```
Pkt1 packet #0.0.0
Pkt1 da:0x00
Pkt1 sa:0xa6
Pkt1 length:0x9b (data.size=155)
Pkt1 data[0]:0x72 data[1]:0x9f .... data[153]:0x53 data[154]:0x4b
```

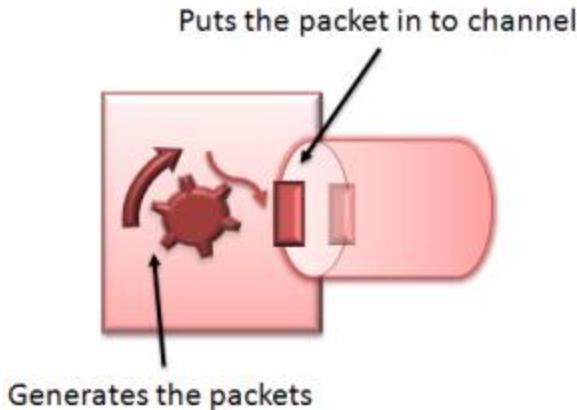
```
Pkt1 fcs:0x24
```

```
Pkt2 packet #0.0.0
Pkt2 da:0x00
Pkt2 sa:0xa6
Pkt2 length:0x9b (data.size=155)
Pkt2 data[0]:0x72 data[1]:0x9f .... data[153]:0x53 data[154]:0x4b
Pkt2 fcs:0x24
```

## **PHASE 5 GENERATOR**

In This phase, we will use the usage of vmm atomic generator. In phase 4, using `vmm\_atomic\_gen macro we defined Packet\_atomic\_gen.

In Environment class, we will create an instance of Packet\_atomic\_gen, and connect it to the instance of Packet\_channel, i.e gen2drv Chan. The Packet\_channel instance gen2drv Chan will be used to connect the atomic generator to the driver instance. We will also configure the atomic generator to generate 10 instances of packet.



© www.testbench.in

1) In Environment class, declare a Packet\_atomic\_gen instance.

```
Packet_atomic_gen atomic_gen;
```

2) Declare a Packet\_channel instance.

```
Packet_channel gen2drv Chan;
```

3) Construct the gen2drv Chan in build() method.

```
gen2drv Chan = new("gen2drv","chan");
```

- 4) Construct the atomic\_gen in build() method.

```
atomic_gen = new("atomic_gen",0,gen2drv Chan);
```

- 5) Configure the atomic\_gen to generate 10 instances.

```
atomic_gen.stop_after_n_insts = 10;
```

- 6) In start() method, start the atomic\_gen.

```
atomic_gen.start_xactor();
```

- 7) In stop() method, call the stop\_xactor() method of atomic\_gen.

```
atomic_gen.stop_xactor();
```

### Environment Class Source Code:

```
`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends vmm_env;

    virtual mem_interface.MEM    mem_intf    ;
    virtual input_interface.IP   input_intf   ;
    virtual output_interface.OP  output_intf[4] ;

    Packet_atomic_gen atomic_gen;
    Packet_channel gen2drv Chan;

function new(virtual mem_interface.MEM    mem_intf_new    ,
            virtual input_interface.IP   input_intf_new  ,
            virtual output_interface.OP  output_intf_new[4] );
    super.new("Environment ");
    this.mem_intf    = mem_intf_new    ;
    this.input_intf  = input_intf_new ;
    this.output_intf = output_intf_new ;
```

```

`vmm_note(this.log, "Created env object");
endfunction : new

virtual function void gen_cfg();
    super.gen_cfg();
    `vmm_note(this.log,"Start of gen_cfg() method ");
    `vmm_note(this.log,"End of gen_cfg() method ");
endfunction

virtual function void build();
    super.build();
    `vmm_note(this.log,"Start of build() method ");

    gen2drv Chan = new("gen2drv","chan");
    atomic_gen = new("atomic_gen",0,gen2drv Chan);
    atomic_gen.stop_after_n_insts = 10;

    `vmm_note(this.log,"End of build() method ");
endfunction

virtual task reset_dut();
    super.reset_dut();
    `vmm_note(this.log,"Start of reset_dut() method ");
    mem_intf.cb.mem_data    <= 0;
    mem_intf.cb.mem_add     <= 0;
    mem_intf.cb.mem_en      <= 0;
    mem_intf.cb.mem_rd_wr   <= 0;
    input_intf.cb.data_in    <= 0;
    input_intf.cb.data_status <= 0;
    output_intf[0].cb.read   <= 0;
    output_intf[1].cb.read   <= 0;
    output_intf[2].cb.read   <= 0;
    output_intf[3].cb.read   <= 0;

    // Reset the DUT
    input_intf.reset    <= 1;
    repeat (4) @ input_intf.clock;
    input_intf.reset    <= 0;

```

```

`vmm_note(this.log,"End of reset_dut() method ");
endtask

virtual task cfg_dut();
    super.cfg_dut();
    `vmm_note(this.log,"Start of cfg_dut() method ");
    mem_intf.cb.mem_en <= 1;
    @(posedge mem_intf.clock);
    mem_intf.cb.mem_rd_wr <= 1;

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h0;
    mem_intf.cb.mem_data <= `P0;
    `vmm_note(this.log ,$psprintf(" Port 0 Address %h ",`P0));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h1;
    mem_intf.cb.mem_data <= `P1;
    `vmm_note(this.log ,$psprintf(" Port 1 Address %h ",`P1));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h2;
    mem_intf.cb.mem_data <= `P2;
    `vmm_note(this.log ,$psprintf(" Port 2 Address %h ",`P2));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h3;
    mem_intf.cb.mem_data <= `P3;
    `vmm_note(this.log ,$psprintf(" Port 3 Address %h ",`P3));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_en <=0;
    mem_intf.cb.mem_rd_wr <= 0;
    mem_intf.cb.mem_add <= 0;
    mem_intf.cb.mem_data <= 0;

`vmm_note(this.log,"End of cfg_dut() method ");
endtask

virtual task start();

```

```

super.start();
`vmm_note(this.log,"Start of start() method ");

atomic_gen.start_xactor();

`vmm_note(this.log,"End of start() method ");
endtask

virtual task wait_for_end();
super.wait_for_end();
`vmm_note(this.log,"Start of wait_for_end() method ");
repeat(1000) @(input_intf.clock);
`vmm_note(this.log,"End of wait_for_end() method ");
endtask

virtual task stop();
super.stop();
`vmm_note(this.log,"Start of stop() method ");

atomic_gen.stop_xactor();

`vmm_note(this.log,"End of stop() method ");
endtask

virtual task cleanup();
super.cleanup();
`vmm_note(this.log,"Start of cleanup() method ");
`vmm_note(this.log,"End of cleanup() method ");
endtask

virtual task report();
`vmm_note(this.log,"Start of report() method \n\n\n");
$display("-----");
super.report();
$display("-----");
$display("\n\n");
`vmm_note(this.log,"End of report() method");
endtask

endclass

```

`endif

### Download the phase 5 source code.

[vmm\\_switch\\_5.tar](#)

[Browse the code in vmm\\_switch\\_5.tar](#)

### Run the simulation

vcs -sverilog -f filelist -R -ntb\_opts rvm

### Log file report:

```
***** Start of testcase *****
Normal[NOTE] on Environment() at      0:
    Created env object
Normal[NOTE] on Environment() at      0:
    Start of gen_cfg() method
Normal[NOTE] on Environment() at      0:
    End of gen_cfg() method
Normal[NOTE] on Environment() at      0:
    Start of build() method
Normal[NOTE] on Environment() at      0:
    End of build() method
Normal[NOTE] on Environment() at      0:
    Start of reset_dut() method
Normal[NOTE] on Environemnt() at      60:
    End of reset_dut() method
Normal[NOTE] on Environemnt() at      60:
    Start of cfg_dut() method
Normal[NOTE] on Environemnt() at      90:
    Port 0 Address 00
Normal[NOTE] on Environemnt() at     110:
    Port 1 Address 11
Normal[NOTE] on Environemnt() at     130:
    Port 2 Address 22
Normal[NOTE] on Environemnt() at     150:
    Port 3 Address 33
Normal[NOTE] on Environemnt() at     170:
    End of cfg_dut() method
Normal[NOTE] on Environemnt() at     170:
```

Start of start() method	
Normal[NOTE] on Environemnt() at	170:
End of start() method	
Normal[NOTE] on Environemnt() at	170:
Start of wait_for_end() method	
Normal[NOTE] on Environemnt() at	10170:
End of wait_for_end() method	
Normal[NOTE] on Environemnt() at	10170:
Start of stop() method	
Normal[NOTE] on Environemnt() at	10170:
End of stop() method	
Normal[NOTE] on Environemnt() at	10170:
Start of cleanup() method	
Normal[NOTE] on Environemnt() at	10170:
End of cleanup() method	
Normal[NOTE] on Environemnt() at	10170:
Start of report() method	

---

Simulation PASSED on ./ (./) at	10170 (0 warnings, 0 demoted errors & 0 demoted warnings)
---------------------------------	---

---

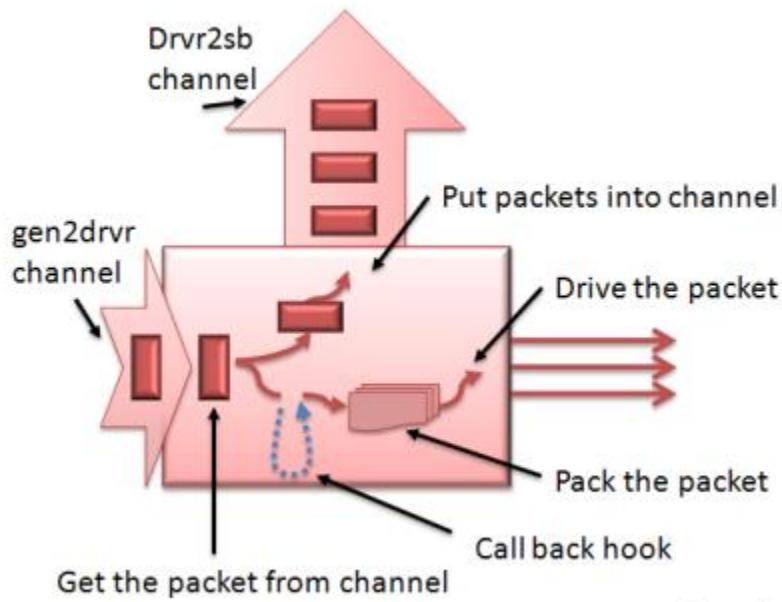
Normal[NOTE] on Environemnt() at	10170:
End of report() method	
*****	***** End of testcase *****

## **PHASE 6 DRIVER**

In phase 6 we will write a driver and then insatiate the driver in environment and send packet in to DUT. Driver class is defined in Driver.sv file.

In this Driver class, take the packets from the generator and then drives it to the DUT input interface and then send the packet to a channel for scoreboard purpose.

In this class, we also add 2 callback methods.



© www.testbench.in

- 1) Extend vmm\_xactor\_callbacks to define Driver\_callbacks. In this class, declare 2 methods pre\_trans() and post\_trans(). pre\_trans() method is called before driving the packet transaction and post\_trans() method will be called after driving the packet transaction.

```

class Driver_callbacks extends vmm_xactor_callbacks;

    // Called before a transaction is executed
    virtual task pre_trans(Packet tr);
        endtask: pre_trans

    // Called after a transaction has been executed
    virtual task post_trans(Packet tr);
        endtask: post_trans

endclass:Driver_callbacks

```

- 1) Extend vmm\_xactor to define Driver class.

```

class Driver extends vmm_xactor;

```

- 2) Declare a virtual input\_interface of the switch. We will connect this to the Physical interface of the top module same as what we did in environment class.

```
virtual input_interface.IP input_intf;
```

3) Define a channel "gen2drv\_chan" which is used to get packets from generator.

```
Packet_channel gen2drv_chan;
```

4) Define a channel "drv2sb\_chan" which is used to send the packets to the score board.

```
Packet_channel drv2sb_chan;
```

4) Define new constructor with arguments, virtual input interface and channels "gen2drv\_chan" and "drv2sb\_chan".

In the constructor, call the parent constructor and pass the instance name and stream\_id.

Connect the channel and virtual interfaces which are passed as constructor arguments to the class members.

```
function new(string inst,  
    int stream_id = -1,  
    virtual input_interface.IP input_intf_new,  
    Packet_channel gen2drv_chan = null,  
    Packet_channel drv2sb_chan = null);  
  
super.new("driver",inst,stream_id);  
  
this.input_intf = input_intf_new;  
  
if(gen2drv_chan == null)  
    `vmm_fatal(log,"gen2drv_channel is null");  
else  
    this.gen2drv_chan = gen2drv_chan;  
  
if(drv2sb_chan == null)  
    `vmm_fatal(log,"drv2sb_channel is null");  
else  
    this.drv2sb_chan = drv2sb_chan;  
  
`vmm_note(log,"Driver created ");  
endfunction
```

4) Define drive() method. This method drives the packet to the dut.

```
task drive(Packet pkt);
    logic [7:0] pack[];
    int pkt_len;

    pkt_len = pkt.byte_pack(pack,0,0);
    @(posedge input_intf.clock);

    for (int i=0;i< pkt_len - 1;i++)
        begin
            @(posedge input_intf.clock);
            input_intf.cb.data_status <= 1 ;
            input_intf.cb.data_in <= pack[i];
        end

    @(input_intf.clock);
    input_intf.cb.data_status <= 0 ;
    input_intf.cb.data_in <= pack[pkt_len -1];
    @(input_intf.clock);
    this.drv2sb_chan.put(pkt);
endtask
```

4) Define the main() method. First call the super.main() method.

```
super.main();
`vmm_note(this.log," started main task");
```

5) In main() method, start a forever thread, which gets the packets from the 'gen2drv\_chan'. The thread iteration has to be block if the channel is empty or stopped.

```
forever begin
    Packet pkt;

    wait_if_stopped_or_empty(this.gen2drv_chan);
    this.gen2drv_chan.get(pkt);

    `vmm_trace(this.log, "Starting transaction...");
    `vmm_debug(this.log, pkt.psdisplay(" "));
```

6) Call the drive() method, which drives the packet to DUT. Call the pre\_tans() callback method using `vmm\_callback macro before calling the drive method and post\_trans() callback method after driving the packet.

```

`vmm_callback(Driver_callbacks,pre_trans(pkt));

drive(pkt);

```

```

`vmm_callback(Driver_callbacks,post_trans(pkt));

```

### Driver Class Source Code:

```

`ifndef GUARD_DRIVER
`define GUARD_DRIVER

```

```

class Driver_callbacks extends vmm_xactor_callbacks;

```

```

// Called before a transaction is executed

```

```

virtual task pre_trans(Packet tr);
endtask: pre_trans

```

```

// Called after a transaction has been executed

```

```

virtual task post_trans(Packet tr);
endtask: post_trans

```

```

endclass:Driver_callbacks

```

```

class Driver extends vmm_xactor;

```

```

virtual input_interface.IP input_intf;
Packet_channel      gen2drv_chan;
Packet_channel      drv2sb_chan;

```

```

function new(string inst,
            int stream_id = -1,
            virtual input_interface.IP input_intf_new,
            Packet_channel  gen2drv_chan = null,
            Packet_channel  drv2sb_chan = null);

```

```

super.new("driver",inst,stream_id);

```

```

this.input_intf = input_intf_new;

```

```

if(gen2drv_chan == null)

```

```

`vmm_fatal(log,"gen2drv_channel is null");

```

```

else

```

```

this.gen2drv_chan = gen2drv_chan;

```

```

if(drv2sb_chan == null)

```

```

`vmm_fatal(log,"drv2sb_channel is null");

```

```

else
    this.drv2sb_chan = drv2sb_chan;

    `vmm_note(log,"Driver created ");
endfunction

task drive(Packet pkt);
    logic [7:0] pack[];
    int pkt_len;

    pkt_len = pkt.byte_pack(pack,0,0);
    @(posedge input_intf.clock);

    for (int i=0;i< pkt_len - 1;i++)
        begin
            @(posedge input_intf.clock);
            input_intf.cb.data_status <= 1 ;
            input_intf.cb.data_in <= pack[i];
        end

        @(input_intf.clock);
        input_intf.cb.data_status <= 0 ;
        input_intf.cb.data_in <= pack[pkt_len -1];
        @(input_intf.clock);
        this.drv2sb_chan.put(pkt);
    endtask

task main();
    super.main();
    `vmm_note(this.log," started main task ");

    forever begin
        Packet pkt;

        wait_if_stopped_or_empty(this.gen2drv_chan);
        this.gen2drv_chan.get(pkt);

        `vmm_trace(this.log, "Starting transaction...");
        `vmm_debug(this.log, pkt.psdisplay(" "));

        `vmm_callback(Driver_callbacks,pre_trans(pkt));
    
```

```

drive(pkt);

`vmm_callback(Driver_callbacks,post_trans(pkt));

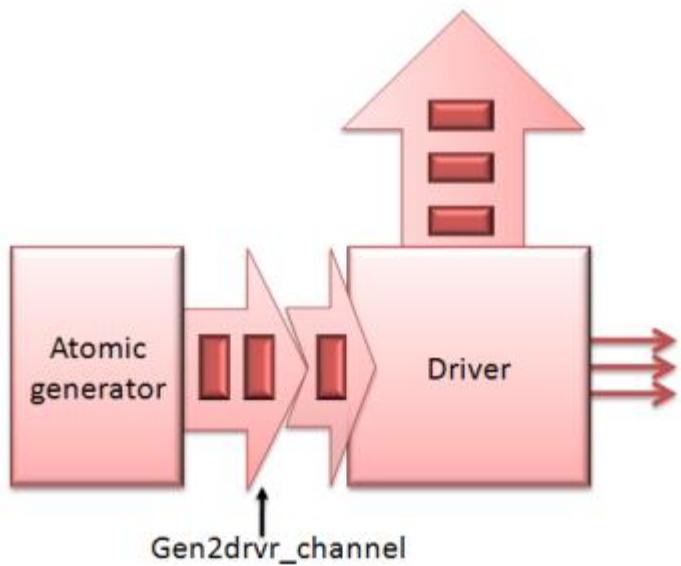
`vmm_trace(this.log, "Completed transaction...");
`vmm_debug(this.log, pkt.psdisplay("  "));
@(posedge input_intf.clock);

end
endtask

endclass
`endif

```

Now we will take the instance of the driver in the environment class.



© www.testbench.in

- 1) Declare a channel "drvrv2sb\_chan" which will be used to connect the scoreboard and driver.

```
Packet_channel drvrv2sb_chan;
```

2) Declare a driver object "drvrv".

Driver drvr;

3) In build method, construct the channel.

drvrv2sb\_chan = new("drvrv2sb","chan");

4) In build method, construct the driver object. Pass the input\_intf and drvrv2sb\_chan channel and gen2drvrv Chan channel.

drvrv = new("Drvrv",0,input\_intf,gen2drvrv Chan,drvrv2sb Chan);

5) To start sending the packets to the DUT, call the start method of "drvrv" in the start method of Environment class.

drvrv.start();

5) In the stop() method, call the stop\_xactor() method of drvr.

drvrv.stop\_xactor();

### **Environment Class Source Code:**

```
`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends vmm_env;

    virtual mem_interface.MEM    mem_intf    ;
    virtual input_interface.IP   input_intf   ;
    virtual output_interface.OP  output_intf[4] ;
```

Packet\_atomic\_gen atomic\_gen;

Driver drrv;

Packet\_channel gen2drvrv Chan;

```

Packet_channel drvr2sb_chan;

function new(virtual mem_interface.MEM  mem_intf_new  ,
            virtual input_interface.IP  input_intf_new  ,
            virtual output_interface.OP output_intf_new[4] );
super.new("Environment ");
this.mem_intf  = mem_intf_new ;
this.input_intf = input_intf_new ;
this.output_intf = output_intf_new ;

`vmm_note(this.log, "Created env object");
endfunction : new

```

```

virtual function void gen_cfg();
    super.gen_cfg();
    `vmm_note(this.log,"Start of gen_cfg() method ");
    `vmm_note(this.log,"End of gen_cfg() method ");
endfunction

virtual function void build();
    super.build();
    `vmm_note(this.log,"Start of build() method ");
    gen2drvr_chan = new("gen2drvr","chan");

    drvr2sb_chan = new("drvr2sb","chan");
    drvr = new("Drvr",0,input_intf,gen2drvr_chan,drvr2sb_chan);

    atomic_gen = new("atomic_gen",0,gen2drvr_chan);
    atomic_gen.stop_after_n_insts = 10;
    `vmm_note(this.log,"End of build() method ");
endfunction

virtual task reset_dut();
    super.reset_dut();
    `vmm_note(this.log,"Start of reset_dut() method ");
    mem_intf.cb.mem_data    <= 0;
    mem_intf.cb.mem_add     <= 0;
    mem_intf.cb.mem_en      <= 0;
    mem_intf.cb.mem_rd_wr   <= 0;
    input_intf.cb.data_in   <= 0;

```

```

input_intf.cb.data_status <= 0;
output_intf[0].cb.read    <= 0;
output_intf[1].cb.read    <= 0;
output_intf[2].cb.read    <= 0;
output_intf[3].cb.read    <= 0;

// Reset the DUT
input_intf.reset      <= 1;
repeat (4) @ input_intf.clock;
input_intf.reset      <= 0;

`vmm_note(this.log,"End of reset_dut() method ");
endtask

virtual task cfg_dut();
super.cfg_dut();
`vmm_note(this.log,"Start of cfg_dut() method ");
mem_intf.cb.mem_en <= 1;
@(posedge mem_intf.clock);
mem_intf.cb.mem_rd_wr <= 1;

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h0;
mem_intf.cb.mem_data <= `P0;
`vmm_note(this.log ,\$psprintf(" Port 0 Address %h ",`P0));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h1;
mem_intf.cb.mem_data <= `P1;
`vmm_note(this.log ,\$psprintf(" Port 1 Address %h ",`P1));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h2;
mem_intf.cb.mem_data <= `P2;
`vmm_note(this.log ,\$psprintf(" Port 2 Address %h ",`P2));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h3;
mem_intf.cb.mem_data <= `P3;
`vmm_note(this.log ,\$psprintf(" Port 3 Address %h ",`P3));

```

```

@(posedge mem_intf.clock);
mem_intf.cb.mem_en  <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add  <= 0;
mem_intf.cb.mem_data <= 0;

`vmm_note(this.log,"End of cfg_dut() method ");
endtask

virtual task start();
super.start();
`vmm_note(this.log,"Start of start() method ");
atomic_gen.start_xactor();

drvr.start_xactor(); 

`vmm_note(this.log,"End of start() method ");
endtask

virtual task wait_for_end();
super.wait_for_end();
`vmm_note(this.log,"Start of wait_for_end() method ");
repeat(1000) @(input_intf.clock);
`vmm_note(this.log,"End of wait_for_end() method ");
endtask

virtual task stop();
super.stop();
`vmm_note(this.log,"Start of stop() method ");
atomic_gen.stop_xactor();

drvr.stop_xactor(); 

`vmm_note(this.log,"End of stop() method ");
endtask

virtual task cleanup();
super.cleanup();
`vmm_note(this.log,"Start of cleanup() method ");

```

```

`vmm_note(this.log,"End of cleanup() method ");
endtask

virtual task report();
    `vmm_note(this.log,"Start of report() method \n\n\n");
    $display("-----");
    super.report();
    $display("-----");
    $display("\n\n");
    `vmm_note(this.log,"End of report() method");
endtask

endclass
`endif

```

**Download the phase 6 source code:**

[vmm\\_switch\\_6.tar](#)

[Browse the code in vmm\\_switch\\_6.tar](#)

**Run the command:**

vcs -sverilog -f filelist -R -ntb\_opts rvm

**Log file report.**

```

***** Start of testcase *****
Normal[NOTE] on Environemnt() at      0:
    Created env object
Normal[NOTE] on Environemnt() at      0:
    Start of gen_cfg() method
Normal[NOTE] on Environemnt() at      0:
    End of gen_cfg() method
Normal[NOTE] on Environemnt() at      0:
    Start of build() method
Normal[NOTE] on driver(Drvr) at      0:
    Driver created
Normal[NOTE] on Environemnt() at      0:
    End of build() method
Normal[NOTE] on Environemnt() at      0:
    Start of reset_dut() method

```

Normal[NOTE] on Environment() at	60:
End of reset_dut() method	
Normal[NOTE] on Environment() at	60:
Start of cfg_dut() method	
Normal[NOTE] on Environment() at	90:
Port 0 Address 00	
Normal[NOTE] on Environment() at	110:
Port 1 Address 11	
Normal[NOTE] on Environment() at	130:
Port 2 Address 22	
Normal[NOTE] on Environment() at	150:
Port 3 Address 33	
Normal[NOTE] on Environment() at	170:
End of cfg_dut() method	
Normal[NOTE] on Environment() at	170:
Start of start() method	
Normal[NOTE] on Environment() at	170:
End of start() method	
Normal[NOTE] on Environment() at	170:
Start of wait_for_end() method	
Normal[NOTE] on driver(Drvr) at	170:
started main task	
Normal[NOTE] on Environment() at	10170:
End of wait_for_end() method	
Normal[NOTE] on Environment() at	10170:
Start of stop() method	
Normal[NOTE] on Environment() at	10170:
End of stop() method	
Normal[NOTE] on Environment() at	10170:
Start of cleanup() method	
Normal[NOTE] on Environment() at	10170:
End of cleanup() method	
Normal[NOTE] on Environment() at	10170:
Start of report() method	

---

Simulation PASSED on ./ (./) at 10170 (0 warnings, 0 demoted errors & 0 demoted warnings)

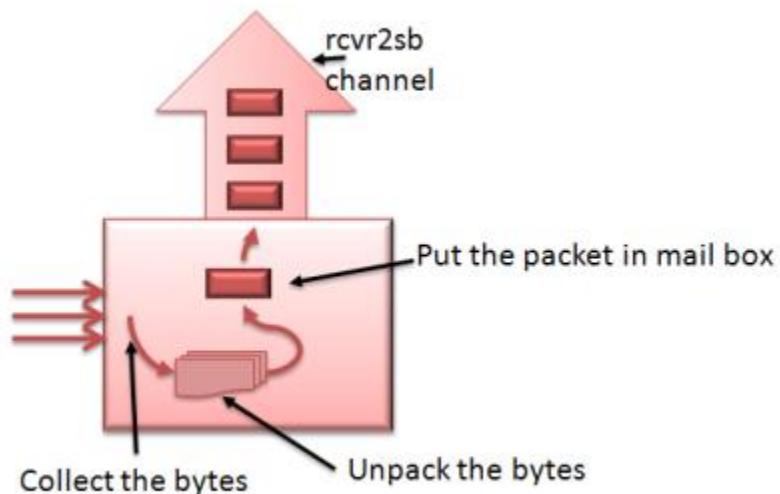
Normal[NOTE] on Environment() at 10170:  
End of report() method  
\*\*\*\*\* End of testcase \*\*\*\*\*

## **PHASE 7 RECEIVER**

In this phase, we will write a receiver and use the receiver in environment class to collect the packets coming from the switch output\_interface.

Receiver collects the data bytes from the interface signal. And then unpacks the bytes into packet and pushes it into channel for score boarding.

Receiver class is written in Receiver.sv file.



© www.testbench.in

- 1) Extend vmm\_xactor and define Receiver class.

```
class Receiver extends vmm_xactor;
```

- 1) Declare a virtual output\_interface. We will connect this to the Physical interface of the top module, same as what we did in environment class.

```
virtual output_interface.OP output_intf;
```

- 2) Declare a channel "rcvr2sb\_chan" which is used to send the packets to the score board

```
Packet_channel rcvr2sb_chan;
```

- 3) Define new constructor with arguments, virtual input interface and a channel which is used to send packets from the receiver to scoreboard. Implement the rest of the logic as it was done in the driver constructor.

```
function new(string inst = "class",
    int unsigned stream_id = -1,
    virtual output_interface.OP output_intf_new,
    Packet_channel rcvr2sb_chan);

super.new("Receiver",inst,stream_id);

this.output_intf = output_intf_new ;

if(rcvr2sb_chan == null)
    `vmm_fatal(log,"rcvr2sb_channel is null");
else
    this.rcvr2sb_chan = rcvr2sb_chan;

`vmm_note(log,"Receiver created ");

endfunction : new
```

- 4) Define the main() method.

In start method, do the following

First call the super.main() method.

Then start a thread which collects the data from the outputinterface and then unpack the data to a packet and put into channel.

```
forever
begin
    repeat(2) @(posedge output_intf.clock);
```

```

wait(output_intf.cb.ready)
output_intf.cb.read <= 1;
repeat(2) @(posedge output_intf.clock);

while (output_intf.cb.ready)
begin
    bytes = new[bytes.size + 1](bytes);
    bytes[bytes.size - 1] = output_intf.cb.data_out;
    @(posedge output_intf.clock);
end

bytes[bytes.size - 1] = output_intf.cb.data_out;
output_intf.cb.read <= 0;
@(posedge output_intf.clock);
`vmm_note(this.log,"Received a packet ");
pkt = new();
pkt.byte_unpack(bytes);
pkt.display("rcvr");

rcvr2sb_chan.put(pkt);

bytes.delete();
end

```

### Receiver Class Source Code:

```

`ifndef GUARD_RECEIVER
`define GUARD_RECEIVER

class Receiver extends vmm_xactor;

virtual output_interface.OP output_intf;
Packet_channel rcvr2sb_chan;

function new(string inst = "class",
            int unsigned stream_id = -1,
            virtual output_interface.OP output_intf_new,
            Packet_channel rcvr2sb_chan);

super.new("Receiver",inst,stream_id);

```

```

this.output_intf = output_intf_new ;

if(rcvr2sb_chan == null)
    `vmm_fatal(log,"rcvr2sb_channel is null");
else
    this.rcvr2sb_chan = rcvr2sb_chan;

`vmm_note(log,"Receiver created ");

endfunction : new

task main();
logic [7:0] bytes[];
Packet pkt;

super.main();
`vmm_note(this.log," started main task ");

forever
begin
    repeat(2) @(posedge output_intf.clock);
    wait(output_intf.cb.ready)
    output_intf.cb.read <= 1;
    repeat(2) @(posedge output_intf.clock);

    while (output_intf.cb.ready)
    begin
        bytes = new[bytes.size + 1](bytes);
        bytes[bytes.size - 1] = output_intf.cb.data_out;
        @(posedge output_intf.clock);
    end

    bytes[bytes.size - 1] = output_intf.cb.data_out;
    output_intf.cb.read <= 0;
    @(posedge output_intf.clock);
    `vmm_note(this.log,"Received a packet ");
    pkt = new();
    pkt.byte_unpack(bytes);

```

```

pkt.display("rcvr");

rcvr2sb_chan.put(pkt);

bytes.delete();

end

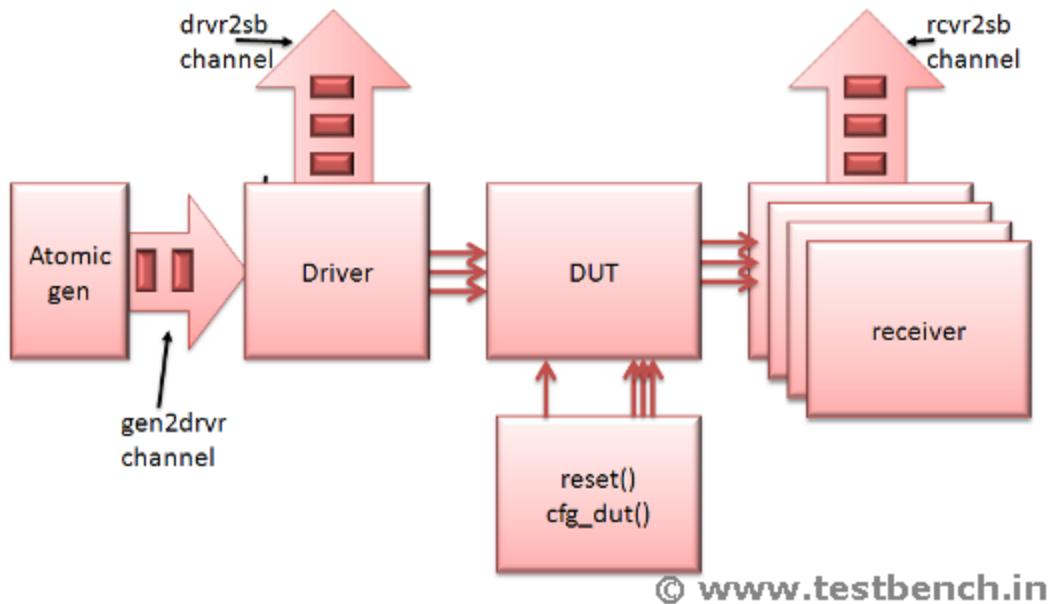
endtask : main

endclass

`endif

```

Now we will take the instance of the receiver in the environment class.



- 1) Declare a channel "rcvr2sb\_chan" which will be used to connect the scoreboard and receiver.

```
Packet_channel rcvr2sb_chan;
```

- 2) Declare 4 receiver object "rcvr".

```
Receiver rcvr[4];
```

3) In build method, construct the rcvr2sb\_chan.

```
rcvr2sb_chan = new("rcvr2sb","chan");
```

4) In build method, construct the receiver object. Pass the output\_intf and rcvr2sb\_chan. There are 4 output interfaces and receiver objects. We will connect one receiver for one output interface.

```
foreach(rcvr[i])
    rcvr[i] = new($psprintf("Rcvr-%0d",i),i,output_intf[i],rcvr2sb_chan);
```

5) To start the receiver activities, call the start\_xactor() method of rcvr objects in the start() method of Environment class.

```
rcvr[0].start_xactor();
rcvr[1].start_xactor();
rcvr[2].start_xactor();
rcvr[3].start_xactor();
```

6) Call the stop\_xactor() method of the receiver object in the stop() method of the Environment .

```
rcvr[0].stop_xactor();
rcvr[1].stop_xactor();
rcvr[2].stop_xactor();
rcvr[3].stop_xactor();
```

### Environment Class Source Code:

```
`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends vmm_env;

    virtual mem_interface.MEM    mem_intf    ;
    virtual input_interface.IP   input_intf   ;
    virtual output_interface.OP  output_intf[4] ;

    Packet_atomic_gen atomic_gen;
```

```

Driver drvr;

Receiver rcvr[4];

Packet_channel gen2drvr_chan;
Packet_channel drvr2sb_chan;

Packet_channel rcvr2sb_chan;

function new(virtual mem_interface.MEM mem_intf_new ,
            virtual input_interface.IP input_intf_new ,
            virtual output_interface.OP output_intf_new[4] );
super.new("Environment ");
this.mem_intf = mem_intf_new ;
this.input_intf = input_intf_new ;
this.output_intf = output_intf_new ;

`vmm_note(this.log, "Created env object");
endfunction : new

virtual function void gen_cfg();
super.gen_cfg();
`vmm_note(this.log,"Start of gen_cfg() method ");
`vmm_note(this.log,"End of gen_cfg() method ");
endfunction

virtual function void build();
super.build();
`vmm_note(this.log,"Start of build() method ");
gen2drvr_chan = new("gen2drvr","chan");
drvr2sb_chan = new("drvr2sb","chan");

rcvr2sb_chan = new("rcvr2sb","chan");

atomic_gen = new("atomic_gen",0,gen2drvr_chan);
atomic_gen.stop_after_n_insts = 10;
drvr = new("Drvr",0,input_intf,gen2drvr_chan,drvr2sb_chan);

foreach(rcvr[i])

```

```

rcvr[i] = new($psprintf("Rcvr-%0d",i),i,output_intf[i],rcvr2sb_chan);

`vmm_note(this.log,"End of build() method ");
endfunction

virtual task reset_dut();
    super.reset_dut();
    `vmm_note(this.log,"Start of reset_dut() method ");
    mem_intf.cb.mem_data    <= 0;
    mem_intf.cb.mem_add     <= 0;
    mem_intf.cb.mem_en      <= 0;
    mem_intf.cb.mem_rd_wr   <= 0;
    input_intf.cb.data_in   <= 0;
    input_intf.cb.data_status <= 0;
    output_intf[0].cb.read   <= 0;
    output_intf[1].cb.read   <= 0;
    output_intf[2].cb.read   <= 0;
    output_intf[3].cb.read   <= 0;

    // Reset the DUT
    input_intf.reset    <= 1;
    repeat (4) @ input_intf.clock;
    input_intf.reset    <= 0;

`vmm_note(this.log,"End of reset_dut() method ");
endtask

virtual task cfg_dut();
    super.cfg_dut();
    `vmm_note(this.log,"Start of cfg_dut() method ");
    mem_intf.cb.mem_en <= 1;
    @(posedge mem_intf.clock);
    mem_intf.cb.mem_rd_wr <= 1;

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h0;
    mem_intf.cb.mem_data <= `P0;
    `vmm_note(this.log ,$psprintf(" Port 0 Address %h ",`P0));

    @(posedge mem_intf.clock);

```

```

mem_intf.cb.mem_add <= 8'h1;
mem_intf.cb.mem_data <= `P1;
`vmm_note(this.log ,\$psprintf(" Port 1 Address %h ",`P1));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h2;
mem_intf.cb.mem_data <= `P2;
`vmm_note(this.log ,\$psprintf(" Port 2 Address %h ",`P2));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h3;
mem_intf.cb.mem_data <= `P3;
`vmm_note(this.log ,\$psprintf(" Port 3 Address %h ",`P3));

@(posedge mem_intf.clock);
mem_intf.cb.mem_en <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add <= 0;
mem_intf.cb.mem_data <= 0;

`vmm_note(this.log,"End of cfg_dut() method ");
endtask

```

```

virtual task start();
super.start();
`vmm_note(this.log,"Start of start() method ");
atomic_gen.start_xactor();
drv.start_xactor();

```

```

rcvr[0].start_xactor();
rcvr[1].start_xactor();
rcvr[2].start_xactor();
rcvr[3].start_xactor();

```

```

`vmm_note(this.log,"End of start() method ");
endtask

```

```

virtual task wait_for_end();
super.wait_for_end();
`vmm_note(this.log,"Start of wait_for_end() method ");

```

```

repeat(1000) @ (input_intf.clock);
`vmm_note(this.log,"End of wait_for_end() method ");
endtask

virtual task stop();
super.stop();
`vmm_note(this.log,"Start of stop() method ");
atomic_gen.stop_xactor();
drv.stop_xactor();

rcvr[0].stop_xactor();
rcvr[1].stop_xactor();
rcvr[2].stop_xactor();
rcvr[3].stop_xactor();

`vmm_note(this.log,"End of stop() method ");
endtask

virtual task cleanup();
super.cleanup();
`vmm_note(this.log,"Start of cleanup() method ");
`vmm_note(this.log,"End of cleanup() method ");
endtask

virtual task report();
`vmm_note(this.log,"Start of report() method \n\n\n");
$display("-----");
super.report();
$display("-----");
$display("\n\n");
`vmm_note(this.log,"End of report() method");
endtask

endclass
`endif

```

**Download the phase 7 source code:**

[vmm\\_switch\\_7.tar](#)

[Browse the code in vmm\\_switch\\_7.tar](#)

**Run the command:**

vcs -sverilog -f filelist -R -ntb\_opts rvm

**Log file report.**

```
***** Start of testcase *****
Normal[NOTE] on Environemnt() at      0:
    Created env object
Normal[NOTE] on Environemnt() at      0:
    Start of gen_cfg() method
Normal[NOTE] on Environemnt() at      0:
    End of gen_cfg() method
Normal[NOTE] on Environemnt() at      0:
    Start of build() method
Normal[NOTE] on driver(Drvr) at      0:
    Driver created
Normal[NOTE] on Receiver(Rcvr-0) at   0:
    Receiver created
Normal[NOTE] on Receiver(Rcvr-1) at   0:
    Receiver created
Normal[NOTE] on Receiver(Rcvr-2) at   0:
    Receiver created
Normal[NOTE] on Receiver(Rcvr-3) at   0:
    Receiver created
Normal[NOTE] on Environemnt() at      0:
    End of build() method
Normal[NOTE] on Environemnt() at      0:
    Start of reset_dut() method
Normal[NOTE] on Environemnt() at     60:
    End of reset_dut() method
Normal[NOTE] on Environemnt() at     60:
    Start of cfg_dut() method
Normal[NOTE] on Environemnt() at     90:
    Port 0 Address 00
Normal[NOTE] on Environemnt() at    110:
    Port 1 Address 11
Normal[NOTE] on Environemnt() at    130:
    Port 2 Address 22
Normal[NOTE] on Environemnt() at    150:
```

Port 3 Address 33

Normal[NOTE] on Environemnt() at 170:  
End of cfg\_dut() method

Normal[NOTE] on Environemnt() at 170:  
Start of start() method

Normal[NOTE] on Environemnt() at 170:  
End of start() method

Normal[NOTE] on Environemnt() at 170:  
Start of wait\_for\_end() method

Normal[NOTE] on driver(Drvr) at 170:  
started main task

Normal[NOTE] on Receiver(Rcvr-0) at 170:  
started main task

Normal[NOTE] on Receiver(Rcvr-1) at 170:  
started main task

Normal[NOTE] on Receiver(Rcvr-2) at 170:  
started main task

Normal[NOTE] on Receiver(Rcvr-3) at 170:  
started main task

Normal[NOTE] on Receiver(Rcvr-0) at 470:  
Received a packet  
rcvr packet #0.0.0  
rcvr da:0x00  
rcvr sa:0xac  
rcvr length:0x05 (data.size=4)  
rcvr data[0]:0xcb data[1]:0x7e data[2]:0x52 data[3]:0xa4  
rcvr fcs:0x29

Normal[NOTE] on Receiver(Rcvr-1) at 710:  
Received a packet  
rcvr packet #0.0.0  
rcvr da:0x11  
rcvr sa:0xf3  
rcvr length:0x06 (data.size=5)  
rcvr data[0]:0xc4 data[1]:0xd5 .... data[3]:0xf3 data[4]:0x88  
rcvr fcs:0x5b

Normal[NOTE] on Receiver(Rcvr-3) at 890:  
Received a packet  
rcvr packet #0.0.0

```
rcvr da:0x33
rcvr sa:0x6b
rcvr length:0x03 (data.size=2)
rcvr data[0]:0x32 data[1]:0x27
rcvr fcs:0x1c
```

Normal[NOTE] on Environemnt() at	10170:
End of wait_for_end() method	
Normal[NOTE] on Environemnt() at	10170:
Start of stop() method	
Normal[NOTE] on Environemnt() at	10170:
End of stop() method	
Normal[NOTE] on Environemnt() at	10170:
Start of cleanup() method	
Normal[NOTE] on Environemnt() at	10170:
End of cleanup() method	
Normal[NOTE] on Environemnt() at	10170:
Start of report() method	

---

Simulation PASSED on ./ (./) at 10170 (0 warnings, 0 demoted errors & 0 demoted warnings)

---

Normal[NOTE] on Environemnt() at 10170:  
    End of report() method  
\*\*\*\*\* End of testcase \*\*\*\*\*

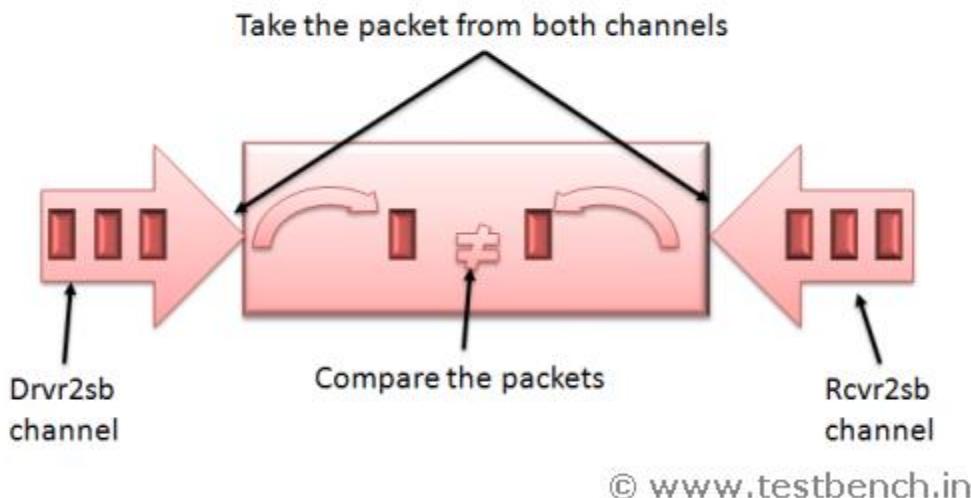
## **PHASE 8 SCOREBOARD**

In this phase we will see the scoreboard implementation. Vmm has scoreboard classes with lot of features. For this example, we will write a simple scoreboard which is implemented using the vmm\_xactor.

Scoreboard has 2 channels. One is used to for getting the packets from the driver and other from

the receiver. Then the packets are compared and if they don't match, then error is asserted. For comparison, compare () method of the Packet class is used.

Scoreboard is implemented in file Scoreboard.sv.



- 1) Declare 2 channels drvr2sb\_chan and rcvr2sb\_chan.

```
Packet_channel drvr2sb_chan;
Packet_channel rcvr2sb_chan;
```

- 2) Declare a constructor method with drvr2sb\_chan , rcvr2sb\_chan , a string for instance name and stream\_id as arguments.

```
function new(string inst = "class",
            int unsigned stream_id = -1,
            Packet_channel drvr2sb_chan = null,
            Packet_channel rcvr2sb_chan = null);
```

- 3) Call the super.new() method and Connect the channels of the constructor to the channels of the scoreboard.

```
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 ");

```

4) Define vmm\_xactor main() method. First call the super.main() method and then do the following steps forever.

Wait until there is a packet in "rcvr2sb\_chan". Then pop the packet from channel.

```

rcvr2sb_chan.get(pkt_rcv);
$display(" %0d : Scorebooard : Scoreboard received a packet from receiver ",$time);

```

Then pop the packet from drvr2sb\_chan.

```
drvr2sb_chan.get(pkt_exp);
```

Compare both packets and assert an error if the comparison fails using `vmm\_error.

```

if(pkt_rcv.compare(pkt_exp,msg))
$display(" %0d : Scoreboard :Packet Matched ",$time);
else
`vmm_error(this.log,$psprintf(" Packet MissMatched \n %s ",msg));

```

### **Scoreboard Class Source Code:**

```

`ifndef GUARD_SCOREBOARD
`define GUARD_SCOREBOARD

```

```
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**

```

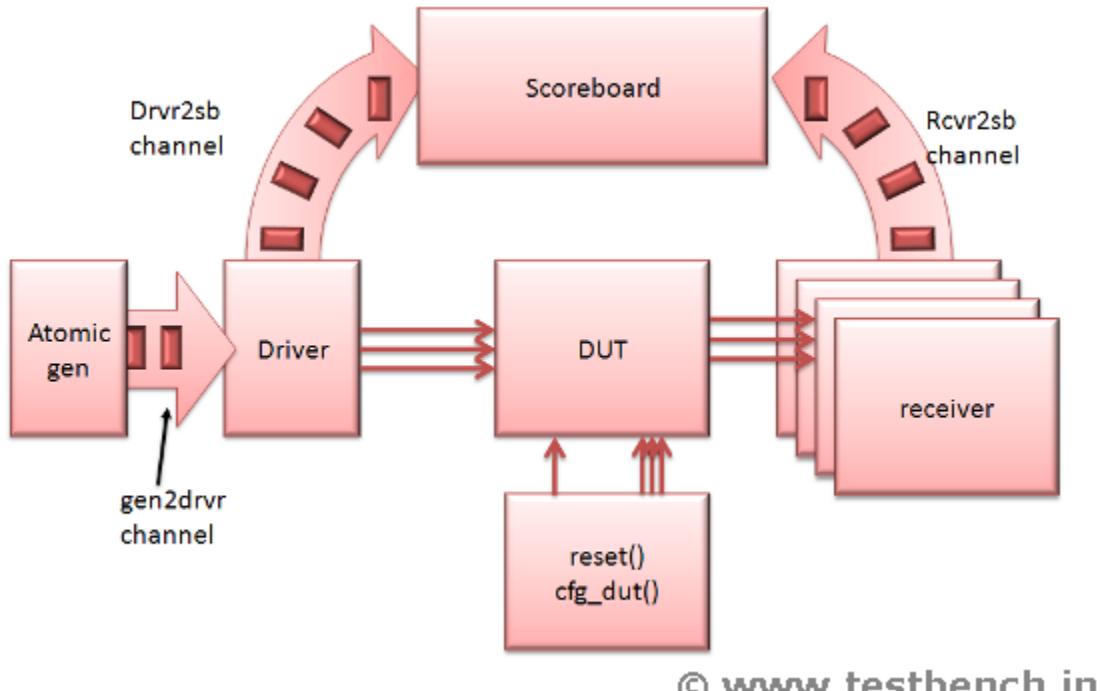
task main();
    Packet pkt_rcv,pkt_exp;
    string msg;
    super.main();
    forever
        begin
            rcvr2sb_chan.get(pkt_rcv);
            $display("%0d : Scoreboard : Scoreboard received a packet from receiver ",$time);
            drvr2sb_chan.get(pkt_exp);
            if(pkt_rcv.compare(pkt_exp,msg))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

endclass

```

```
`endif
```

Now we will see how to connect the scoreboard in the Environment class.



© www.testbench.in

- 1) Declare a scoreboard handle.

```
Scoreboard sb;
```

- 2) Construct the scoreboard in the build() method. Pass the drvr2sb\_chan and rcvr2sb\_chan channels to the score board constructor.

```
sb = new("Sb",0,drvrv2sb_chan,rcvr2sb_chan);
```

- 3) Start the scoreboard activities in the start() method.

```
sb.start_xactor();
```

- 4) Stop the scoreboard activities in stop() method.

```
sb.stop_xactor();
```

### **Source Code Of The Environment Class:**

```
`ifndef GUARD_ENV
`define GUARD_ENV

class Environment extends vmm_env;

    virtual mem_interface.MEM    mem_intf    ;
    virtual input_interface.IP   input_intf   ;
    virtual output_interface.OP  output_intf[4] ;

    Packet_atomic_gen atomic_gen;
    Driver drvr;
    Receiver rcvr[4];

    Scoreboard sb;

    Packet_channel gen2drvr_chan;
    Packet_channel drvr2sb_chan;
    Packet_channel rcvr2sb_chan;

function new(virtual mem_interface.MEM    mem_intf_new    ,
            virtual input_interface.IP   input_intf_new   ,
            virtual output_interface.OP  output_intf_new[4] );
    super.new("Environment ");
    this.mem_intf    = mem_intf_new    ;
    this.input_intf   = input_intf_new ;
    this.output_intf  = output_intf_new ;

    `vmm_note(this.log, "Created env object");
endfunction : new

virtual function void gen_cfg();
    super.gen_cfg();
    `vmm_note(this.log,"Start of gen_cfg() method ");
    `vmm_note(this.log,"End of gen_cfg() method ");
```

```

endfunction

virtual function void build();
    super.build();
    `vmm_note(this.log,"Start of build() method ");
    gen2drv Chan = new("gen2drv","chan");
    drv2sb Chan = new("drv2sb","chan");
    rcvr2sb Chan = new("rcvr2sb","chan");
    atomic_gen = new("atomic_gen",0,gen2drv Chan);
    atomic_gen.stop_after_n_insts = 10;
    drvrv = new("Drvrv",0,input_intf,gen2drv Chan,drv2sb Chan);
    foreach(rcvr[i])
        rcvr[i] = new($psprintf("Rcvr-%0d",i),i,output_intf[i],rcvr2sb Chan);

    sb = new("Sb",0,drv2sb Chan,rcvr2sb Chan);

    `vmm_note(this.log,"End of build() method ");
endfunction

virtual task reset_dut();
    super.reset_dut();
    `vmm_note(this.log,"Start of reset_dut() method ");
    mem_intf.cb.mem_data    <= 0;
    mem_intf.cb.mem_add     <= 0;
    mem_intf.cb.mem_en      <= 0;
    mem_intf.cb.mem_rd_wr   <= 0;
    input_intf.cb.data_in    <= 0;
    input_intf.cb.data_status <= 0;
    output_intf[0].cb.read   <= 0;
    output_intf[1].cb.read   <= 0;
    output_intf[2].cb.read   <= 0;
    output_intf[3].cb.read   <= 0;

    // Reset the DUT
    input_intf.reset    <= 1;
    repeat (4) @ input_intf.clock;
    input_intf.reset    <= 0;

    `vmm_note(this.log,"End of reset_dut() method ");
endtask

```

```

virtual task cfg_dut();
    super.cfg_dut();
    `vmm_note(this.log,"Start of cfg_dut() method ");
    mem_intf.cb.mem_en <= 1;
    @(posedge mem_intf.clock);
    mem_intf.cb.mem_rd_wr <= 1;

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h0;
    mem_intf.cb.mem_data <= `P0;
    `vmm_note(this.log ,\$psprintf(" Port 0 Address %h ",`P0));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h1;
    mem_intf.cb.mem_data <= `P1;
    `vmm_note(this.log ,\$psprintf(" Port 1 Address %h ",`P1));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h2;
    mem_intf.cb.mem_data <= `P2;
    `vmm_note(this.log ,\$psprintf(" Port 2 Address %h ",`P2));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_add <= 8'h3;
    mem_intf.cb.mem_data <= `P3;
    `vmm_note(this.log ,\$psprintf(" Port 3 Address %h ",`P3));

    @(posedge mem_intf.clock);
    mem_intf.cb.mem_en <=0;
    mem_intf.cb.mem_rd_wr <= 0;
    mem_intf.cb.mem_add <= 0;
    mem_intf.cb.mem_data <= 0;

    `vmm_note(this.log,"End of cfg_dut() method ");
endtask

virtual task start();
    super.start();
    `vmm_note(this.log,"Start of start() method ");

```

```

atomic_gen.start_xactor();
drvr.start_xactor();
rcvr[0].start_xactor();
rcvr[1].start_xactor();
rcvr[2].start_xactor();
rcvr[3].start_xactor();

sb.start_xactor();

`vmm_note(this.log,"End of start() method ");
endtask

virtual task wait_for_end();
    super.wait_for_end();
    `vmm_note(this.log,"Start of wait_for_end() method ");
    repeat(1000) @(input_intf.clock);
    `vmm_note(this.log,"End of wait_for_end() method ");
endtask

virtual task stop();
    super.stop();
    `vmm_note(this.log,"Start of stop() method ");
    atomic_gen.stop_xactor();
    drvr.stop_xactor();
    rcvr[0].stop_xactor();
    rcvr[1].stop_xactor();
    rcvr[2].stop_xactor();
    rcvr[3].stop_xactor();

sb.stop_xactor();

`vmm_note(this.log,"End of stop() method ");
endtask

virtual task cleanup();
    super.cleanup();
    `vmm_note(this.log,"Start of cleanup() method ");
    `vmm_note(this.log,"End of cleanup() method ");
endtask

```

```

virtual task report();
    `vmm_note(this.log,"Start of report() method \n\n");
    $display("-----");
    super.report();
    $display("-----");
    $display("\n\n");
    `vmm_note(this.log,"End of report() method");
endtask

endclass
`endif

```

**Download the phase 8 score code:**

[vmm\\_switch\\_8.tar](#)

Browse the code in [vmm\\_switch\\_8.tar](#)

**Run the simulation:**

`vcs -sverilog -f filelist -R -ntb_opts rvm`

**Log File Report:**

```

***** Start of testcase *****
Normal[NOTE] on Environemnt() at      0:
    Created env object
Normal[NOTE] on Environemnt() at      0:
    Start of gen_cfg() method
Normal[NOTE] on Environemnt() at      0:
    End of gen_cfg() method
Normal[NOTE] on Environemnt() at      0:
    Start of build() method
Normal[NOTE] on driver(Drvr) at      0:
    Driver created
Normal[NOTE] on Receiver(Rcvr-0) at   0:
    Receiver created
Normal[NOTE] on Receiver(Rcvr-1) at   0:
    Receiver created
Normal[NOTE] on Receiver(Rcvr-2) at   0:
    Receiver created

```

Normal[NOTE] on Receiver(Rcvr-3) at 0:  
    Receiver created

Normal[NOTE] on sb(Sb) at 0:  
    Scoreboard created

Normal[NOTE] on Environemnt() at 0:  
    End of build() method

Normal[NOTE] on Environemnt() at 0:  
    Start of reset\_dut() method

Normal[NOTE] on Environemnt() at 60:  
    End of reset\_dut() method

Normal[NOTE] on Environemnt() at 60:  
    Start of cfg\_dut() method

Normal[NOTE] on Environemnt() at 90:  
    Port 0 Address 00

Normal[NOTE] on Environemnt() at 110:  
    Port 1 Address 11

Normal[NOTE] on Environemnt() at 130:  
    Port 2 Address 22

Normal[NOTE] on Environemnt() at 150:  
    Port 3 Address 33

Normal[NOTE] on Environemnt() at 170:  
    End of cfg\_dut() method

Normal[NOTE] on Environemnt() at 170:  
    Start of start() method

Normal[NOTE] on Environemnt() at 170:  
    End of start() method

Normal[NOTE] on Environemnt() at 170:  
    Start of wait\_for\_end() method

Normal[NOTE] on driver(Drvr) at 170:  
    started main task

Normal[NOTE] on Receiver(Rcvr-0) at 170:  
    started main task

Normal[NOTE] on Receiver(Rcvr-1) at 170:  
    started main task

Normal[NOTE] on Receiver(Rcvr-2) at 170:  
    started main task

Normal[NOTE] on Receiver(Rcvr-3) at 170:  
    started main task

size 8 \*\*\*\*

Normal[NOTE] on Receiver(Rcvr-0) at 470:

Received a packet  
rcvr packet #0.0.0  
rcvr da:0x00  
rcvr sa:0xac  
rcvr length:0x05 (data.size=4)  
rcvr data[0]:0xcb data[1]:0x7e data[2]:0x52 data[3]:0xa4  
rcvr fcs:0x29

470 : Scoreboard : Scoreboard received a packet from receiver  
size 9 \*\*\*\*

Normal[NOTE] on Receiver(Rcvr-1) at 710:

Received a packet  
rcvr packet #0.0.0  
rcvr da:0x11  
rcvr sa:0xf3  
rcvr length:0x06 (data.size=5)  
rcvr data[0]:0xc4 data[1]:0xd5 .... data[3]:0xf3 data[4]:0x88  
rcvr fcs:0x5b

size 6 \*\*\*\*

Normal[NOTE] on Receiver(Rcvr-3) at 890:

Received a packet  
rcvr packet #0.0.0  
rcvr da:0x33  
rcvr sa:0x6b  
rcvr length:0x03 (data.size=2)  
rcvr data[0]:0x32 data[1]:0x27  
rcvr fcs:0x1c

Normal[NOTE] on Environment() at 10170:

End of wait\_for\_end() method

Normal[NOTE] on Environment() at 10170:

Start of stop() method

Normal[NOTE] on Environment() at 10170:

End of stop() method

Normal[NOTE] on Environment() at 10170:

Start of cleanup() method

Normal[NOTE] on Environment() at 10170:

End of cleanup() method

Normal[NOTE] on Environment() at 10170:

Start of report() method

---

Simulation PASSED on ./ (./) at 10170 (0 warnings, 0 demoted errors & 0 demoted warnings)

---

Normal[NOTE] on Environemnt() at 10170:

End of report() method

\*\*\*\*\* End of testcase \*\*\*\*\*

## **PHASE 9 COVERAGE**

In this phase we will write the functional coverage for switch protocol. Functional coverage is written in DrvrCovCallback.sv file. After running simulation, you will analyze the coverage results and find out if some test scenarios have not been exercised and write tests to exercise them.

The points which we need to cover are

- 1) Cover all the port address configurations.
- 2) Cover all the packet lengths.
- 3) Cover all correct and incorrect length fields.
- 4) Cover good and bad FCS.
- 5) Cover all the above combinations.

Feature	Done
length	✓
da	✗
....	✓
.....	✗

© www.tutorialspoint.in

We will do the coverage sampling in the driver porst\_trans() callback method which we developed in PHASE\_6.

1) Define a cover group with following cover points. Define this cover group in a class which extends Driver\_callbacks.

```
class DrvrCovCallback extends Driver_callbacks;
```

a) All packet lengths:

```
length : coverpoint pkt.length;
```

b) All port address:

```
da    : coverpoint pkt.da {  
    bins p0 = { `P0 };  
    bins p1 = { `P1 };  
    bins p2 = { `P2 };  
    bins p3 = { `P3 }; }
```

c) Correct and incorrect Length field types:

```
length_kind : coverpoint pkt.length_kind;
```

d) Good and Bad FCS:

```
fcs_kind : coverpoint pkt.fcs_kind;
```

5) Cross product of all the above cover points:

```
all_cross: cross length,da,length_kind,fcs_kind;
```

2) In constructor method, construct the cover group

```
function new();  
    switch_coverage = new();  
endfunction : new
```

3) Write task which calls the sample method to cover the points.

```
task sample(packet pkt);  
    this(pkt) = pkt;  
    switch_coverage.sample();
```

**endtask**:sample

**Source Code Of Coverage Class:**

```
`ifndef GUARD_DRVR_CALLBACK_1
`define GUARD_DRVR_CALLBACK_1

class DrvrCovCallback extends Driver_callbacks;
    Packet pkt;

    covergroup switch_coverage;
        length : coverpoint pkt.length;
        da    : coverpoint pkt.da {
            bins p0 = { `P0 };
            bins p1 = { `P1 };
            bins p2 = { `P2 };
            bins p3 = { `P3 }; }
        length_kind : coverpoint pkt.length_kind;
        fcs_kind : coverpoint pkt.fcs_kind;
        all_cross: cross length,da,length_kind,fcs_kind;
    endgroup

    function new();
        switch_coverage = new();
    endfunction : new

    virtual task post_trans(Packet pkt);
        this(pkt = pkt;
        switch_coverage.sample());
    endtask: post_trans

endclass:DrvrCovCallback

`endif
```

Now we will connect this callback instance to the Driver.

1) Take the instance of DrvrCovCallback in the Environment class.

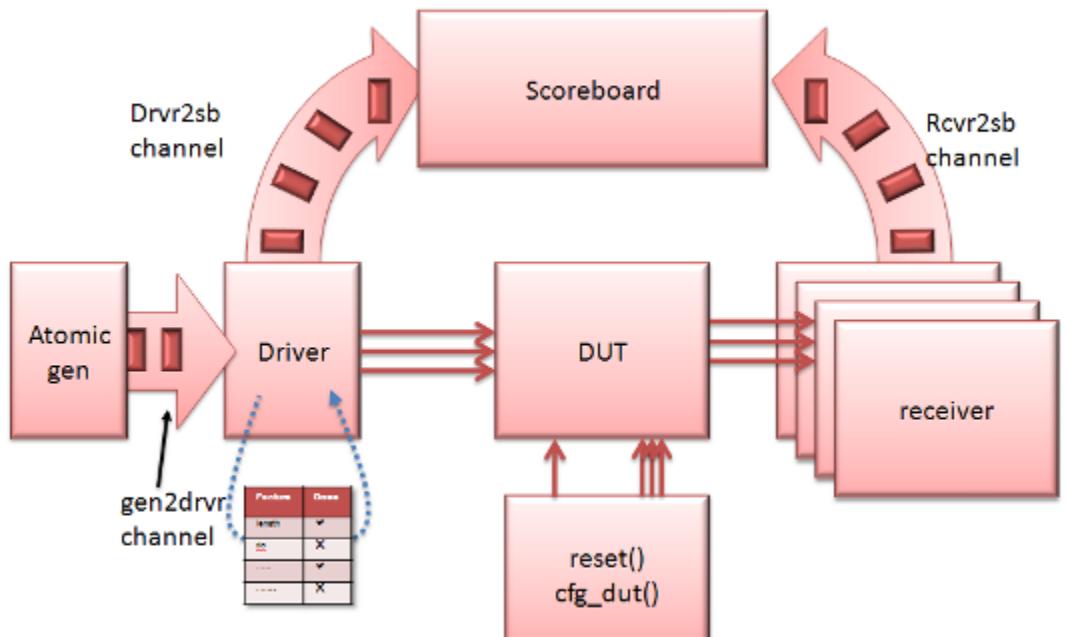
```
DrvrvCovCallback cov_cb;
```

2) Construct the cov\_cb object in build method.

```
cov_cb = new();
```

3) In the Build method, call the append\_callback() of the drvr and pass the cov\_cb object.

```
drvr.append_callback(cov_cb);
```



© www.testbench.in

Source code of the Environment class:

```
`ifndef GUARD_ENV  
`define GUARD_ENV
```

```

class Environment extends vmm_env;

    virtual mem_interface.MEM    mem_intf    ;
    virtual input_interface.IP   input_intf   ;
    virtual output_interface.OP  output_intf[4] ;

    Packet_atomic_gen atomic_gen;
    Driver drvr;
    Receiver rcvr[4];
    Scoreboard sb;

    DrvrCovCallback cov_cb;

    Packet_channel gen2drvr_chan;
    Packet_channel drvr2sb_chan;
    Packet_channel rcvr2sb_chan;

function new(virtual mem_interface.MEM    mem_intf_new    ,
            virtual input_interface.IP   input_intf_new   ,
            virtual output_interface.OP  output_intf_new[4] );
    super.new("Environment ");
    this.mem_intf    = mem_intf_new    ;
    this.input_intf   = input_intf_new ;
    this.output_intf  = output_intf_new ;
    `vmm_note(this.log, "Created env object");
endfunction : new

virtual function void gen_cfg();
    super.gen_cfg();
    `vmm_note(this.log,"Start of gen_cfg() method ");
    `vmm_note(this.log,"End of gen_cfg() method ");
endfunction

virtual function void build();
    super.build();
    `vmm_note(this.log,"Start of build() method ");
    gen2drvr_chan = new("gen2drvr","chan");

```

```

drvrv2sb_chan = new("drvrv2sb","chan");
rcvr2sb_chan = new("rcvr2sb","chan");
atomic_gen = new("atomic_gen",0,gen2drvrv Chan);
atomic_gen.stop_after_n_insts = 10;
drvrv = new("Drvrv",0,input_intf,gen2drvrv Chan,drvrv2sb Chan);
foreach(rcvr[i])
  rcvr[i] = new($psprintf("Rcvr-%0d",i),i,output_intf[i],rcvr2sb Chan);
  sb = new("Sb",0,drvrv2sb Chan,rcvr2sb Chan);

cov_cb = new();
drvrv.append_callback(cov_cb);

`vmm_note(this.log,"End of build() method ");
endfunction

virtual task reset_dut();
  super.reset_dut();
  `vmm_note(this.log,"Start of reset_dut() method ");
  mem_intf.cb.mem_data    <= 0;
  mem_intf.cb.mem_add     <= 0;
  mem_intf.cb.mem_en      <= 0;
  mem_intf.cb.mem_rd_wr   <= 0;
  input_intf.cb.data_in   <= 0;
  input_intf.cb.data_status <= 0;
  output_intf[0].cb.read   <= 0;
  output_intf[1].cb.read   <= 0;
  output_intf[2].cb.read   <= 0;
  output_intf[3].cb.read   <= 0;

  // Reset the DUT
  input_intf.reset    <= 1;
  repeat (4) @ input_intf.clock;
  input_intf.reset    <= 0;

`vmm_note(this.log,"End of reset_dut() method ");
endtask

virtual task cfg_dut();
  super.cfg_dut();
  `vmm_note(this.log,"Start of cfg_dut() method ");

```

```

mem_intf.cb.mem_en <= 1;
@(posedge mem_intf.clock);
mem_intf.cb.mem_rd_wr <= 1;

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h0;
mem_intf.cb.mem_data <= `P0;
`vmm_note(this.log ,\$psprintf(" Port 0 Address %h ",`P0));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h1;
mem_intf.cb.mem_data <= `P1;
`vmm_note(this.log ,\$psprintf(" Port 1 Address %h ",`P1));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h2;
mem_intf.cb.mem_data <= `P2;
`vmm_note(this.log ,\$psprintf(" Port 2 Address %h ",`P2));

@(posedge mem_intf.clock);
mem_intf.cb.mem_add <= 8'h3;
mem_intf.cb.mem_data <= `P3;
`vmm_note(this.log ,\$psprintf(" Port 3 Address %h ",`P3));

@(posedge mem_intf.clock);
mem_intf.cb.mem_en <=0;
mem_intf.cb.mem_rd_wr <= 0;
mem_intf.cb.mem_add <= 0;
mem_intf.cb.mem_data <= 0;

`vmm_note(this.log,"End of cfg_dut() method ");
endtask

virtual task start();
super.start();
`vmm_note(this.log,"Start of start() method ");
atomic_gen.start_xactor();
drvr.start_xactor();
rcvr[0].start_xactor();
rcvr[1].start_xactor();

```

```

rcvr[2].start_xactor();
rcvr[3].start_xactor();
sb.start_xactor();
`vmm_note(this.log,"End of start() method ");
endtask

virtual task wait_for_end();
    super.wait_for_end();
    `vmm_note(this.log,"Start of wait_for_end() method ");
    repeat(1000) @(input_intf.clock);
    `vmm_note(this.log,"End of wait_for_end() method ");
endtask

virtual task stop();
    super.stop();
    `vmm_note(this.log,"Start of stop() method ");
    `vmm_note(this.log,"End of stop() method ");
endtask

virtual task cleanup();
    super.cleanup();
    `vmm_note(this.log,"Start of cleanup() method ");
    `vmm_note(this.log,"End of cleanup() method ");
endtask

virtual task report();
    `vmm_note(this.log,"Start of report() method \n\n\n");
    $display("-----");
    super.report();
    $display("-----");
    $display("\n\n");
    `vmm_note(this.log,"End of report() method");
endtask

endclass
`endif

```

**Download the phase 9 score code:**

## vmm\_switch\_9.tar

[Browse the code in vmm\\_switch\\_9.tar](#)

### Run the simulation:

```
vcs -sverilog -f filelist -R -ntb_opts rvm  
urg -dir simv.cm/
```

## AVM INTRODUCTION

Mentor Graphics Advanced Verification Methodology(AVM) is opensource and non-proprietary SystemVerilog and systemc methodology. This is based on OSCI(open systemC initiative) TLM standard. AVM can be downloaded for free from mentors website. AVM contains TLM(Transaction Level modeling) concepts, rules, suggestions with lot of examples for better understanding. AVM focuses on constrained randomization, Coverage , Assertions & scorboarding.

AVM provides base classes to improve user productivity. AVM has TLM interfaces to communicate between testbench components. AVM library has the following base classes.

Reporting class: Has reporting methods, verbosity level controls etc.

Building Blocks: `avm_transaction`, `avm_stimulus`, `avm_in_order_comparator`, `avm_env` etc.

TML library: TLM interfaces, TLM channels.

Verification components can be class or modules. Modules are handy for HDL users. One of the main advantage of using module based avm testbench is it supports system verilog assertions.In SystemVerilog ,Assertions cannot be part of class.

Where as class are more flexible as they are OO. Randomization is easy with class. Dynamic instantiation and classes can be used as variables for communication.

AVM components interact with DUT using SystemVerilog virtual interface.

Here I'm going to discuss the avm base classes which are used in the example. For more details refer to avm cookbook.

### Tlm:

All the verification components use TLM interfaces whether they are class based or module

based. Main operations on the TLM or put, get and peek.

Initiator puts transaction to a target and can get transaction from a target.

There are two types of TLMs.

BLOCKING MODEL: Waits until the put transaction is completed before the next put operation.  
Blocking operation is time consuming.

NON BLOCKING MODEL: Nonblocking operations are instantaneous. Every thing happens in zero time.

TLM supports unidirectional and bidirectional data flow. tlm\_fifo are bases for TLM. tlm\_fifo has put,peek(get a copy) and get interface for communication transactions. It also has put\_ap,get\_ap(analyses ports for communicating with score board or coverage model).

Lets see how to use ports and exports.

The TLM interface is called port. tlm\_fifo sub blocks are called export.

## **EXAMPLE**

Declare a tlm\_fifo:

```
tlm_fifo#(packet) gen2drv;
```

Create **put** port in initiator block(lets call it gen):

```
tlm_blocking_put_if#(packet) put_port;
```

Create **get** port in target block(lets call it drvr):

```
tlm_blocking_get_if#(packet) get_port;
```

Connect the exports to ports:

```
gen.put_port = gen2drv.blocking_put_export;  
drvr.get_port = gen2drv.blocking_get_export;
```

Now **put** can be done in gen **and** **get** can be done in drvr.

## **Building Blocks**

There are two types of base classes used for building AVM components.

- 1)avm\_named\_component
- 2)avm\_verificatino\_component.

avm\_named\_components has information about the hierarchy of the instantiation and are used for TLM port and export connections. This also has built in message controlling system.

avm\_verification\_components are inherited from avm\_named\_component.  
avm\_verification\_component are used to build testbench component like driver, monitor, transactor etc. In addition to avm\_named\_component features, a run() method is provided for users to define, and this run() method is forked from avm\_env automatically.

### **Avm transactors:**

Avm transaction is base class for creating transactions. In order to allow the avm and tlm libraries to work, we need methods to print, compare and clone transactions.

For any given transaction, T, to do four things to use avm\_transaction:

Inherit from avm\_transaction

Implement the virtual function string convert2string. This is for messaging.

Implement a function bit comp( input T t ). Used by the comparators for checking in scoreboard.

Implement a function T clone(). This is used for tlm\_fifos. clone() returns a copy() of .this(its object handle).

### **Avm env:**

avm\_env class is the top level of testbench. avm\_env has do\_test() method which starts building and execution of the testbench components.

Environment execution starts when do\_test() task is called.

There are 5 execution phases.

Construct: Constructs all the class based verification components.

Connect: Connect the class based verification components together.

Configure: Configuration of the testbench components in Zero time.

Execute: do\_run() forks off all the components run() methods and then execute method.

Report: End of simulation messages goes here.

### **Avm messaging:**

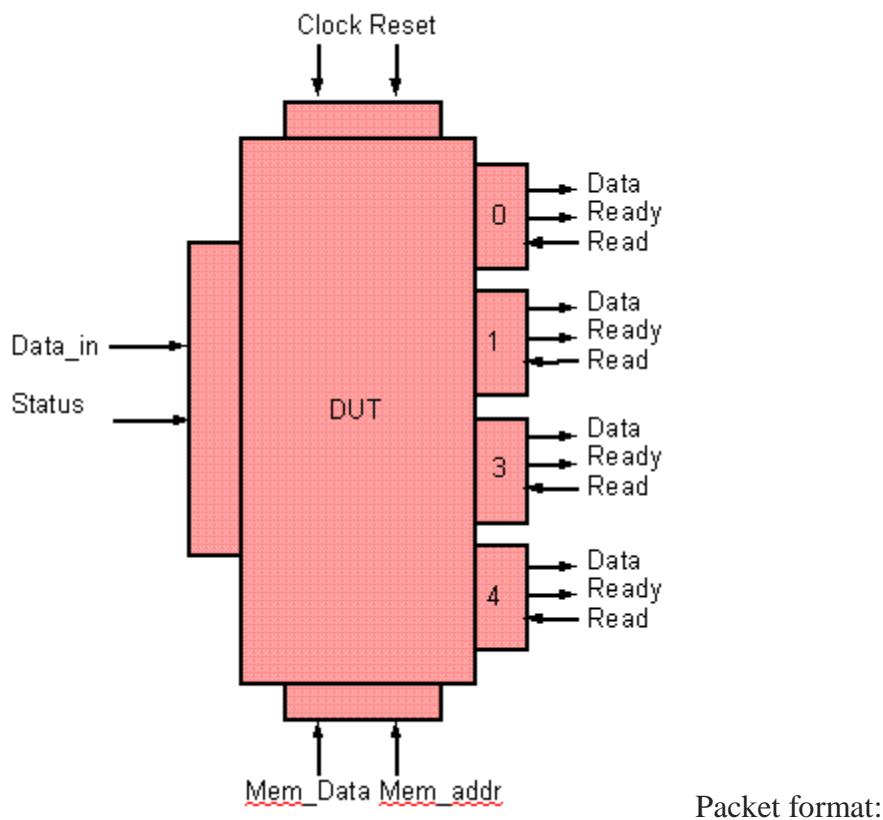
avm\_messageing supports four types of severity levels. They are MESSAGE,WARNING,ERROR,FATAL

avm\_messageing supports four types of actions. They are DISPLAY,LOG,COUNT,EXIT.  
Each of the actions are define by severity and id.

### **DUT SPECIFICATION**

This DUT is a simple switch, which can drive the incoming packet to destination ports based on the address contained in the packet.

The dut contain one input interface from which the packet enters the dut. It has four output interfaces where the packet is driven out.



Packet : Header, data and frame check sequence. Packet width is 8 bits and the length of the packet can be between 4 bytes to 259 bytes.

Packet header:

Packet header contains three fields DA, SA and length.

DA: Destination address of the packet. It is 8 bits. The switch drives the packet to respective ports based on this destination address of the packets.

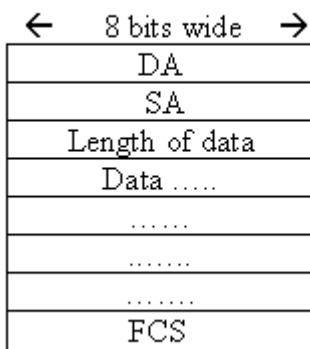
SA: Source address of the packet from where it originate.

Length: This is the length of the data. It can be from 0 to 255.

Data: Data should be in terms of bytes. It can be between 0 to 255 bytes.

FCS: This field contains the security check of the packet. It is calculated over the header and data.

**Packet Structure:**



### **Configuration:**

Dut has four output ports. These output ports have to be configure to a address. Dut matches the DA field of the packet with this configured port address and sends the packet on to that port. To configure the dut, a memory interface is provided. The address of the ports should be unique. It is 8 bits wide. Memory address (0,1,2,3) contains the address of port(0,1,2,4) respectively.

### **Interface Specification:**

The dut has one input Interface, from where the packet enters the dut and 4 output interfaces from where the packet comes out and one memory interface, through the port address can be configured.

### **Memory Interface:**

Through memory interfaced output port address are configured. It accepts 8 bit data to be written to memory. It has 8 bit address inputs. Address 0,1,2,3 contains the address of the port 0,1,2,3 respectively. If the DA feild in the packet matches with the configugred address of any port ,then the packet comes out of that port.

### **Input Interface:**

The status signal has to be high when data is when packet is sent on to the dut it has to become low after sending last byte of the packet. 2 clocks gap should be maintained between packets.

### **Output Interface:**

There are 4 ports, each having data, ready and read signals.

When the data is ready to be sent out from the port, dut makes the ready signal high indicating that data is ready to be sent.

If the read signal is made high when ready is high, then the data comes out of the data signal.

### **RTL**

#### **CODE:rtl.v**

```
module fifo (clk,
    reset,
    write_enb,
    read,
    data_in,
    data_out,
    empty,
    full);
    input clk;
    input reset;
    input write_enb;
    input read;
    input [7:0] data_in;
    output [7:0] data_out;
    output empty;
    output full;
    wire clk;
    wire write_enb;
    wire read;
```

```

wire [7:0] data_in;
reg [7:0] data_out;
wire empty;
wire full;
reg [7:0] ram[0:25];
reg tmp_empty;
reg tmp_full;
integer write_ptr;
integer read_ptr;
always@(negedge reset)
begin
    data_out = 8'b0000_0000;
    tmp_empty = 1'b1;
    tmp_full = 1'b0;
    write_ptr = 0;
    read_ptr = 0;
end

assign empty = tmp_empty;
assign full = tmp_full;
always @(posedge clk) begin
    if ((write_enb == 1'b1) && (tmp_full == 1'b0)) begin
        ram[write_ptr] = data_in;
        tmp_empty <= 1'b0;
        write_ptr = (write_ptr + 1) % 16;
        if (read_ptr == write_ptr) begin
            tmp_full <= 1'b1;
        end
    end
end

if ((read == 1'b1) && (tmp_empty == 1'b0)) begin
    data_out <= ram[read_ptr];
    tmp_full <= 1'b0;
    read_ptr = (read_ptr + 1) % 16;
    if (read_ptr == write_ptr) begin
        tmp_empty <= 1'b1;
    end
end
end
endmodule //fifo

```

```

module port_fsm (clk,
    reset,
    write_enb,
    ffee,
    hold,
    data_status,
    data_in,
    data_out,
    mem0,
    mem1,
    mem2,
    mem3,
    addr);
input clk;
input reset;
input [7:0] mem0;
input [7:0] mem1;
input [7:0] mem2;
input [7:0] mem3;
output[3:0] write_enb;
input ffee;
input hold;
input data_status;
input[7:0] data_in;
output[7:0] data_out;
output [7:0] addr;
reg [7:0] data_out;
reg [7:0] addr;
reg [3:0] write_enb_r;
reg fsm_write_enb;
reg [3:0] state_r;
reg [3:0] state;
reg [7:0] parity;
reg [7:0] parity_delayed;
reg sus_data_in,error;

parameter ADDR_WAIT = 4'b0000;
parameter DATA_LOAD = 4'b0001;
parameter PARITY_LOAD = 4'b0010;

```

```

parameter HOLD_STATE = 4'b0011;
parameter BUSY_STATE = 4'b0100;

always@(negedge reset)
begin
    error      = 1'b0;
    data_out   = 8'b0000_0000;
    addr       = 8'b00000000;
    write_enb_r = 3'b000;
    fsm_write_enb = 1'b0;
    state_r    = 4'b0000;
    state      = 4'b0000;
    parity     = 8'b0000_0000;
    parity_delayed = 8'b0000_0000;
    sus_data_in = 1'b0;
end
assign busy = sus_data_in;
always @(data_status) begin : addr_mux
    if (data_status == 1'b1) begin
        case (data_in)
            mem0 : begin
                write_enb_r[0] = 1'b1;
                write_enb_r[1] = 1'b0;
                write_enb_r[2] = 1'b0;
                write_enb_r[3] = 1'b0;
            end
            mem1 : begin
                write_enb_r[0] = 1'b0;
                write_enb_r[1] = 1'b1;
                write_enb_r[2] = 1'b0;
                write_enb_r[3] = 1'b0;
            end
            mem2 : begin
                write_enb_r[0] = 1'b0;
                write_enb_r[1] = 1'b0;
                write_enb_r[2] = 1'b1;
                write_enb_r[3] = 1'b0;
            end
            mem3 : begin

```

```

write_enb_r[0] = 1'b0;
write_enb_r[1] = 1'b0;
write_enb_r[2] = 1'b0;
write_enb_r[3] = 1'b1;
end
default :write_enb_r = 3'b000;
endcase
// $display(" data_inii %d ,mem0 %d ,mem1 %d ,mem2 %d
mem3",data_in,mem0,mem1,mem2,mem3);
end //if
end //addr_mux;
always @(posedge clk) begin : fsm_state
    state_r <= state;
end //fsm_state;

always @(state_r or data_status or ffee or hold or data_in)
begin : fsm_core
    state = state_r; //Default state assignment
    case (state_r)
        ADDR_WAIT : begin
            if ((data_status == 1'b1) &&
                ((mem0 == data_in)|| (mem1 == data_in)|| (mem3 == data_in) ||(mem2== data_in)))
begin
                if (ffee == 1'b1) begin
                    state = DATA_LOAD;
                end
                else begin
                    state = BUSY_STATE;
                end //if
            end //if;
            sus_data_in = !ffee;
            if ((data_status == 1'b1) &&
                ((mem0 == data_in)|| (mem1 == data_in)|| (mem3 == data_in) ||(mem2== data_in))
&&
                (ffee == 1'b1)) begin
                    addr = data_in;
                    data_out = data_in;
                    fsm_write_enb = 1'b1;
                end
            end
        end
    end

```

```

else begin
    fsm_write_enb = 1'b0;
end //if
end // of case ADDR_WAIT
PARITY_LOAD : begin
    state = ADDR_WAIT;
    data_out = data_in;
    fsm_write_enb = 1'b0;
end // of case PARITY_LOAD
DATA_LOAD : begin
    if ((data_status == 1'b1) &&
        (hold == 1'b0)) begin
        state = DATA_LOAD;
    end
    else if ((data_status == 1'b0) &&
        (hold == 1'b0)) begin
        state = PARITY_LOAD;
    end
    else begin
        state = HOLD_STATE;
    end //if
    sus_data_in = 1'b0;
    if ((data_status == 1'b1) &&
        (hold == 1'b0)) begin
        data_out = data_in;
        fsm_write_enb = 1'b1;
    end
    else if ((data_status == 1'b0) &&
        (hold == 1'b0)) begin
        data_out = data_in;
        fsm_write_enb = 1'b1;
    end
    else begin
        fsm_write_enb = 1'b0;
    end //if
end //end of case DATA_LOAD
HOLD_STATE : begin
    if (hold == 1'b1) begin
        state = HOLD_STATE;
    end

```

```

else if((hold == 1'b0) && (data_status == 1'b0)) begin
    state = PARITY_LOAD;
end
else begin
    state = DATA_LOAD;
end //if
if(hold == 1'b1) begin
    sus_data_in = 1'b1;
    fsm_write_enb = 1'b0;
end
else begin
    fsm_write_enb = 1'b1;
    data_out = data_in;
end //if
end //end of case HOLD_STATE
BUSY_STATE : begin
if(ffee == 1'b0) begin
    state = BUSY_STATE;
end
else begin
    state = DATA_LOAD;
end //if
if(ffee == 1'b0) begin
    sus_data_in = 1'b1;
end
else begin
    addr = data_in; // hans
    data_out = data_in;
    fsm_write_enb = 1'b1;
end //if
end //end of case BUSY_STATE
endcase
end //fsm_core

assign write_enb[0] = write_enb_r[0] & fsm_write_enb;
assign write_enb[1] = write_enb_r[1] & fsm_write_enb;
assign write_enb[2] = write_enb_r[2] & fsm_write_enb;
assign write_enb[3] = write_enb_r[3] & fsm_write_enb;

endmodule //port_fsm

```

```
module switch (clk,
    reset,
    data_status,
    data,
    port0,
    port1,
    port2,
    port3,
    ready_0,
    ready_1,
    ready_2,
    ready_3,
    read_0,
    read_1,
    read_2,
    read_3,
    mem_en,
    mem_rd_wr,
    mem_add,
    mem_data);
    input      clk;
    input      reset;
    input      data_status;
    input [7:0] data;
    input mem_en;
    input mem_rd_wr;
    input [1:0] mem_add;
    input [7:0] mem_data;
    output [7:0] port0;
    output [7:0] port1;
    output [7:0] port2;
    output [7:0] port3;
    output      ready_0;
    output      ready_1;
    output      ready_2;
    output      ready_3;
    input      read_0;
    input      read_1;
    input      read_2;
    input      read_3;
```

```

wire [7:0] data_out_0;
wire [7:0] data_out_1;
wire [7:0] data_out_2;
wire [7:0] data_out_3;
wire ll0;
wire ll1;
wire ll2;
wire ll3;
wire empty_0;
wire empty_1;
wire empty_2;
wire empty_3;
wire ffee;
wire ffee0;
wire ffee1;
wire ffee2;
wire ffee3;
wire ld0;
wire ld1;
wire ld2;
wire ld3;
wire hold;
wire [3:0] write_enb;
wire [7:0] data_out_fsm;
wire [7:0] addr;

reg [7:0]mem[3:0];
wire reset;
fifo queue_0 (.clk    (clk),
              .reset   (reset),
              .write_enb (write_enb[0]),
              .read    (read_0),
              .data_in  (data_out_fsm),
              .data_out (data_out_0),
              .empty    (empty_0),
              .full     (ll0));

fifo queue_1 (.clk    (clk),
              .reset   (reset),
              .write_enb (write_enb[1]),

```

```

.read (read_1),
.data_in (data_out_fsm),
.data_out (data_out_1),
.empty (empty_1),
.full (ll1);
```

```

fifo queue_2 (.clk (clk),
.reset (reset),
.write_enb (write_enb[2]),
.read (read_2),
.data_in (data_out_fsm),
.data_out (data_out_2),
.empty (empty_2),
.full (ll2));
```

```

fifo queue_3 (.clk (clk),
.reset (reset),
.write_enb (write_enb[3]),
.read (read_3),
.data_in (data_out_fsm),
.data_out (data_out_3),
.empty (empty_3),
.full (ll3));
```

```

port_fsm in_port (.clk (clk),
.reset (reset),
.write_enb (write_enb),
.ffee (ffee),
.hold (hold),
.data_status (data_status),
.data_in (data),
.data_out (data_out_fsm),
.mem0 (mem[0]),
.mem1 (mem[1]),
.mem2 (mem[2]),
.mem3 (mem[3]),
.addr (addr));
```

**assign** port0 = data\_out\_0; //make note assignment only for

```

//consistency with vlog env

assign port1 = data_out_1;
assign port2 = data_out_2;
assign port3 = data_out_3;

assign ready_0 = ~empty_0;
assign ready_1 = ~empty_1;
assign ready_2 = ~empty_2;
assign ready_3 = ~empty_3;

assign ffee0 = (empty_0 | (addr != mem[0]));
assign ffee1 = (empty_1 | (addr != mem[1]));
assign ffee2 = (empty_2 | (addr != mem[2]));
assign ffee3 = (empty_3 | (addr != mem[3]));

assign ffee = ffee0 & ffee1 & ffee2 & ffee3;

assign ld0 = (ll0 & (addr == mem[0]));
assign ld1 = (ll1 & (addr == mem[1]));
assign ld2 = (ll2 & (addr == mem[2]));
assign ld3 = (ll3 & (addr == mem[3]));

assign hold = ld0 | ld1 | ld2 | ld3;

always@(posedge clk)
begin

if(mem_en)
if(mem_rd_wr)
begin
mem[mem_add]=mem_data;
///$display("%d %d %d %d",mem_add,mem[0],mem[1],mem[2],mem[3]);
end
end
endmodule //router

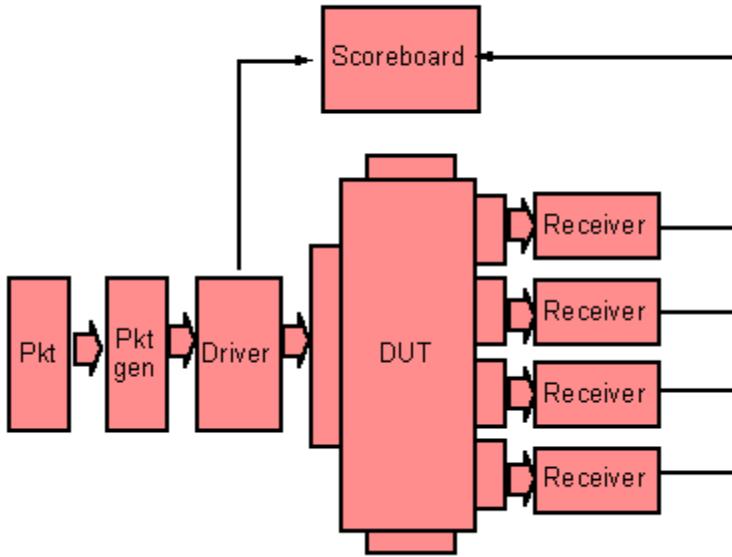
```

## TOP

### Verilog Top

Top level module contains the design and testbench instance. Top module also contains clock

generator. There is no need to instantiate the top module. Testbench and dut instances are connected using interface instance. Make an instance of env class and creat it. Call the do\_test() method which starts the testbench components.



### CODE: top.v

```
`include "mem_env.sv"
```

```
module top();
```

```
// Make an instance of SystemVerilog interface.
```

```
switch_if intf();
```

```
//As the RTL is in verilog and the SV Interface ports are not used,Connect signals by signal names
```

```
switch switch1 (.clk      (intf.clock),
               .reset     (intf.reset),
               .data_status (intf.data_status),
               .data      (intf.data_in),
               port0     (intf.data_out[0]),
               port1     (intf.data_out[1]),
               port2     (intf.data_out[2]),
               port3     (intf.data_out[3]),
```

```

    .ready_0  (intf.ready[0]),
    .ready_1  (intf.ready[1]),
    .ready_2  (intf.ready[2]),
    .ready_3  (intf.ready[3]),
    .read_0   (intf.read[0]),
    .read_1   (intf.read[1]),
    .read_2   (intf.read[2]),
    .read_3   (intf.read[3]),
    .mem_en   (intf.mem_en),
    .mem_rd_wr (intf.mem_rd_wr),
    .mem_add   (intf.mem_add),
    .mem_data   (intf.mem_data));

```

// Create a clock generator.

**initial begin**

intf.clock = 0;

#10;

**forever begin**

#5 intf.clock = !intf.clock;

**end**

**end**

// Make an instance of testbench env

sw\_env env;

**initial**

**begin**

@(posedge intf.clock);

// Pass the interface to testbench environment.

env = new(intf);

// Call do\_test task. Execution of the testbench starts.

env.do\_test;

\$finish;

**end**

**endmodule //top**

## INTERFACE

Create an interface .Define modport to specifies direction. These are used to connect testbench environment and dut in top module.

#### **CODE:interface**

```
`ifndef INTF
`define INTF

interface switch_if();
    bit      data_status;
    bit [7:0] data_in;
    wire [3:0][7:0] data_out;
    wire [3:0] ready;
    bit [3:0] read;
    bit [7:0] mem_data;
    bit [1:0] mem_add;
    bit reset;
    bit mem_en;
    bit mem_rd_wr;
    bit clock;

modport TB(
    output data_status,
    output data_in,
    input data_out,
    input ready,
    output read,
    output mem_data,
    output mem_add,
    output reset,
    output mem_en,
    output mem_rd_wr
);

endinterface:switch_if
`endif
```

#### **ENVIRONMENT**

Environment is extended class of avm\_env. Environment holds all the instances of avm\_verification\_component and fifos for connections.

## CODE:env.sv

```
import avm_pkg::*;

`include "Configuration.sv"
`include "packet.sv"
`include "mem_driver.sv"
`include "reciever.sv"
`include "score_board.sv"

class sw_env extends avm_env;

    // channels and interface
    local virtual switch_if intf;
    tlm_fifo#(packet) gen2drv;
    tlm_fifo#(packet) drv2sb;
    tlm_fifo#(packet) rcv2sb;

    // specific components
    local Configuration cfg;
    local driver drvr;
    local reciever rcvr_0;
    local reciever rcvr_1;
    local reciever rcvr_2;
    local reciever rcvr_3;
    local scoreboard sb;
    local generator gen;
    local packet pkt;

    string msg;

    function new( virtual switch_if intf );
        this.intf = intf;
        cfg = new("cfg",intf);
        pkt = new();
        drvr = new("driver",cfg);
        rcvr_0 = new("reciever_0",0);
    endfunction
endclass
```

```

rcvr_1 = new("reciever_1",1);
rcvr_2 = new("reciever_2",2);
rcvr_3 = new("reciever_3",3);
sb = new();
gen = new();
gen2drv = new();
drv2sb = new();
rcv2sb = new();

endfunction

function void connect();
    avm_report_message("gen_cfg"," Starting... Gen_cfg \n");
    if (!cfg.randomize())
        avm_report_error("gen_cfg","Configuration Randomization Failed!\n");
        cfg.display();
    avm_report_message("gen_cfg"," Ending.... Gen_cfg \n");

    pkt.do_cfg(cfg);
// connect all the virtual interfaces.
drvr.intf = intf;
rcvr_0.intf = intf;
rcvr_1.intf = intf;
rcvr_2.intf = intf;
rcvr_3.intf = intf;

// connect all the TLM interfaces
gen.put_port = gen2drv.blocking_put_export;
drvr.get_port = gen2drv.blocking_get_export;
drvr.put_sb = drv2sb.blocking_put_export;
rcvr_0.put_sb = rcv2sb.blocking_put_export;
sb.drv_port = drv2sb.blocking_get_export;
sb.rcv_port = rcv2sb.blocking_get_export;

endfunction

```

```

task execute;

wait(sb.no_rcv_pkt == 10 );
  terminate;

endtask

task terminate;
  $swrite(msg,"\\n\\n Total number of packets sent %d, Total no of packet received
%d\\n\\n",sb.no_drv_pkt,sb.no_rcv_pkt);
  avm_report_message("env",msg);
endtask

endclass
PACKET

```

Packet is inherited from avm\_trancion. 3 methods copy,comp and convert2string definition. convert2string method returns a string which describes the transaction. comp method is for determining equality of two objects. clone method returns a handle to a newly allocated copy of this(object).

#### **CODE:packet.sv**

```

`ifndef PKT_CLASS
`define PKT_CLASS

//Define the enumerated types for packet payload size type
typedef enum { SMALL_P, MEDIUM_P, LARGE_P } payload_size_t ;
typedef enum { GOOD_P, BAD_P } packet_kind_t;

class packet extends avm_transaction ;

```

```

string msg;

rand payload_size_t payload_size;//Control field for the payload size
byte uid; // Unique id field to identify the packet
rand bit [7:0] len;
rand bit [7:0] da;
rand bit [7:0] sa;

rand byte data[];//Payload using Dynamic array,size is generated on the fly
rand byte parity;
static bit [7:0] mem [3:0];

constraint addr_8bit { (da == mem[3])||(da == mem[0])||(da == mem[1])||(da== mem[2]);}

// Constrain the len according the payload_size control field
constraint len_size {
    (payload_size == SMALL_P ) -> len inside { [5 : 6]};
    (payload_size == MEDIUM_P ) -> len inside { [7 : 8]};
    (payload_size == LARGE_P ) -> len inside {[9 : 10]}; }

// Control field for GOOD/BAD
rand packet_kind_t packet_kind;

// May be assigned either a good or bad value,parity will be calculated in portrandomize
constraint parity_type {
    (packet_kind == GOOD_P ) -> parity == 0;
    (packet_kind == BAD_P ) -> parity != 0; }

// define clone as per avm
function packet clone();
    packet t = new();
    t.copy( this );
    return t;
endfunction

function void do_cfg(Configuration cfg);
    this.mem[0]= cfg.da_port[0];

```

```

this.mem[1]= cfg.da_port[1];
this.mem[2]= cfg.da_port[2];
this.mem[3]= cfg.da_port[3];
$swrite(msg, " packet new ::%x %x %x %x",mem[0],mem[1],mem[2],mem[3]);
avm_report_message("packet",msg);
endfunction

// as per avm, define convert2string
function string convert2string;
  string psdisplay;
  int i;

$write(psdisplay, "da:0x%h sa:0x%h len:0x%h \n",this.da,this.sa,this.len);

for (i = 0; i < this.len; i++) $write(psdisplay, "%s data[%0d] 0x%h \n", psdisplay,i, data[i]);
$write(psdisplay,"%s parity :0x%h \n",psdisplay,this.parity);
convert2string = psdisplay;
endfunction

```

```
function void copy(input packet to = null);
```

```

// Copying to an existing instance. Correct type?
if (!$cast(this, to))
  begin
    avm_report_error("packet", "Attempting to copy to a non packet instance");
    return;
  end

```

```

this.da = to.da;
this.sa = to.sa;
this.len = to.len;
this.data = new[to.len];
foreach(data[i])

```

```

begin
this.data[i] = to.data[i];
end

this.parity = to.parity;
return;

endfunction

//unpacking function for converting received data to class properties
function void unpack(byte bytes[$]);
    $swrite(msg," bytes size %d",bytes.size());
    void'(bytes.pop_front());
    da = bytes.pop_front();
    sa = bytes.pop_front();
    len = bytes.pop_front();

    $swrite(msg,"recieved packet::da:0x%h sa:0x%h len:0x%h
\n", this.da, this.sa, this.len);
    avm_report_message("packet",msg);
    data = new [len - 4];
    parity = bytes.pop_back();
    foreach (data[i]) data[i] = bytes.pop_front();
endfunction

function byte parity_cal();
integer i;
byte result ;
result = result ^ this.da;
result = result ^ this.sa;
result = result ^ this.len;
for (i = 0;i<this.len;i++)
begin
result = result ^ this.data[i];
end
return result;
endfunction

```

```

//post randomize fun to cal parity
function void post_randomize();
    data = new[len];
    foreach(data[i])
        data[i] = $random();
        parity = parity ^ parity_cal();
    endfunction

function void pre_randomize();
    data.delete();
endfunction

// define comp a per avm
function bit comp(input packet cmp,input packet to);
    string diff;
    bit compare;
    compare = 1; // Assume success by default.
    diff = "No differences found";

    if (!$cast(cmp, to))
        begin
            avm_report_error("packet", "Attempting to compare to a non packet instance");
            compare = 0;
            diff = "Cannot compare non packets";

        return compare;
    end

    // data types are the same, do comparison:
    if (to.da != cmp.da)
        begin
            $swrite(diff,"Different DA values: %b != %b", to.da, cmp.da);
            compare = 0;
            avm_report_error("packet",diff);
            return compare;
        end

    if (to.sa != cmp.sa)
        begin

```

```

$swrite(diff,"Different SA values: %b != %b", to.sa, cmp.sa);
compare = 0;
avm_report_error("packet",diff);
return compare;
end
if (to.len != cmp.len)
begin
$swrite(diff,"Different LEN values: %b != %b", to.len, cmp.len);
compare = 0;
avm_report_error("packet",diff);
return compare;
end

foreach(data[i])
if (to.data[i] != cmp.data[i])
begin
$swrite(diff,"Different data[%0d] values: 0x%h != 0x%h",i, to.data[i], cmp.data[i]);
compare = 0;
avm_report_error("packet",diff);
return compare;
end
if (to.parity != cmp.parity)
begin
$swrite(diff,"Different PARITY values: %b != %b", to.parity, cmp.parity);
compare = 0;
avm_report_error("packet",diff);

return compare;
end
return 1;
endfunction

function int unsigned byte_pack(ref logic [7:0] bytes[],
                               input int unsigned offset ,
                               input int kind);
byte_pack = 0;
bytes = new[this.len + 4];
bytes[0] = this.da;
bytes[1] = this.sa;

```

```

bytes[2] = this.len;
foreach(data[i])
bytes[3+i] = data[i];
bytes[this.len + 3 ] = parity;
byte_pack = this.len + 4;
endfunction

endclass
`endif

```

## **PACKET GENERATOR**

Use a class to build packet generator. Make an instance of this class. Packet generator generator generates packets and sends to driver using gen2drv channel. gen2drv is used to connect the packet generator and driver. Packet generator generates the packet and randomizes the packet. Then the packet is put into gen2drv channel. Always check whether the randomization is sucessful and display a message.

**CODE:** gen.sv

```

class generator extends avm_verification_component;

// define tml interface for communication
tlm_blocking_put_if#(packet) put_port;

task run;

packet pkt;

for(int i = 0; i < 10; i++)
begin
pkt = new();

if(!pkt.randomize())
avm_report_error("generator", " randomiation failed\n");
else
avm_report_message("generator", " randomization done\n");
put_port.put(pkt);
end
endtask

```

**endclass**

## **CONFIGURATION**

Configuration class is inherited from `avm_verification_component`. This class generates addresses to configure the dut 4 output ports.

### **CODE:CFG.SV**

```
`ifndef CFG_CLASS
`define CFG_CLASS
class Configuration extends avm_verification_component;
    rand bit [7:0] da_port [4];
    string msg;

    constraint da_ports { (da_port[0] != da_port[1])&&(da_port[1] != da_port[2])&&(da_port[2] !
= da_port[3]);}

    virtual function void display(string prefix = "Test Configuration");
        $swrite(msg," addresss %x %x %x %x \n",da_port[0],da_port[1],da_port[2],da_port[3]);
        avm_report_message("cfg",msg);
    endfunction

    function new(string nm,virtual switch_if intf, avm_named_component p = null);
        super.new(nm,p);
        avm_report_message("cfg"," Configuration Created \n");
    endfunction

    task run;
    endtask

endclass
`endif
```

## **DRIVER**

Driver is a tranctor. It is extension of `avm_verification_component`. Get the packets from `gen2drv` chennel and drive them on to dut interface. Put the packets in to `drv2sb` channel.

## CODE:driver.sv

```
class driver extends avm_verification_component;

virtual switch_if intf;
string msg;
packet pkt;
Configuration cfg;

tlm_blocking_get_if#(packet) get_port;
tlm_blocking_put_if#(packet) put_sb;

function new(string nm, Configuration cfg);
    this.cfg = cfg;

    super.new(nm);
endfunction

function void connect;

endfunction

task run;

    reset_dut();
    cfg_dut();
forever
begin
    get_port.get(pkt);
    $display("consumer: sendging %s packet\n", pkt.convert2string);
    drive(pkt);
    avm_report_message("Driver","Puting packet to score board");
    put_sb.put(pkt);
    @(negedge intf.clock);

end

endtask
```

```

task drive(packet pkt);
logic [7:0] pack[];
int pkt_len;
pkt_len = pkt.byte_pack(pack,0,0);
$swrite(this.msg,"Packed packet length %d \n",pkt_len);
avm_report_message("Driver",this.msg);
@(negedge intf.clock);
for (int i=0;i< pkt_len - 1;i++)
begin
  @(negedge intf.clock);
  intf.data_status <= 1 ;
  intf.data_in <= pack[i];
end
  @(negedge intf.clock);
  intf.data_status <= 0 ;
  intf.data_in <= pack[pkt_len -1];
  @(negedge intf.clock);
endtask

```

```

task reset_dut();
avm_report_message("reset_dut"," Starting... reset_dut \n");
  @(negedge intf.clock);
  avm_report_message("reset_dut"," Starting... reset_dut \n");
  intf.data_status <= 0;
  intf.data_in <= 0;
  intf.read <= 0;
  intf.mem_data <= 0;
  intf.mem_add <= 0;
  intf.reset <= 0;
  intf.mem_en <= 0;
  intf.mem_rd_wr <= 0;
  @(negedge intf.clock);
#2 intf.reset <= 1;
  @(negedge intf.clock);
#2 intf.reset <= 0;
  @(negedge intf.clock);
  @(negedge intf.clock);
avm_report_message("reset_dut"," Ending... reset_dut \n");

```

**endtask**

```
task cfg_dut();  
    avm_report_message("cfg_dut"," Starting... cfg_dut \n");  
    for(int i = 0;i<4 ;i++)  
    begin  
        intf.mem_en  <= 1;  
        @(negedge intf.clock);  
        intf.mem_rd_wr <= 1;  
        @(negedge intf.clock);  
        intf.mem_add  <= i;  
        intf.mem_data <= cfg.da_port[i];  
    end  
    @(negedge intf.clock);  
    intf.mem_en  <= 0;  
    intf.mem_rd_wr <= 0;  
    intf.mem_add  <= 0;  
    intf.mem_data <= 0;  
  
    avm_report_message("cfg_dut"," Ending... cfg_dut \n");
```

**endtask**

**endclass**

## RECIEVER

Reciever is a tranctor. Start the collection of packet from dut in the run() task and send them to score\_board through rcv2sb channel.

**CODE:**reciever.sv

```
class reciever extends avm_verification_component;  
static tlm_blocking_put_if#(packet) put_sb;  
virtual switch_if intf;  
int port;  
string name;  
function new(string nm, int port);  
    super.new(nm);  
    this.name =nm;  
    this.port = port;
```

```

endfunction

task run;
    byte received_bytes[$] ;
    packet recv_pkt,pkt;
    pkt = new();
    forever
        begin
            @(posedge (intf.ready[port]));
            while (intf.ready[port]) begin
                intf.read <= 4'b0001 << port;
                @(negedge intf.clock);
                received_bytes.push_back(intf.data_out[port]);
            end
            intf.read <= 4'h0;

            pkt.unpack(received_bytes);
            received_bytes = {};
            recv_pkt = new pkt;
            put_sb.put(recv_pkt);
        end
    endtask
endclass

```

## SCOREBOARD

When the packet comes from receiver, the scoreboard gets the expected packet from drv2sb channel and compare them.

### CODE:scoreboard.sv

```

`ifndef SB_CLASS
`define SB_CLASS

class scoreboard extends avm_verification_component;

```

```

packet exp_pkt,drv_pkt;
packet exp_que[$];
packet recv_pkt;

integer no_drv_pkt;
integer no_recv_pkt;
string msg;
tlm_blocking_get_if#(packet) drv_port;
tlm_blocking_get_if#(packet) recv_port;

function new();
super.new("Scoreboard");

no_drv_pkt = 0;
no_recv_pkt = 0;

avm_report_message("sb","Scoreboard created");
endfunction

task run();
avm_report_message("sb"," STARTED main task ");
fork
forever
begin
drv_port.get(drv_pkt);
this.no_drv_pkt++;
$swrite(msg,"Received packet no from driver %d size of
queue%d\n",no_recv_pkt,exp_que.size());
avm_report_message("sb",msg);
exp_que.push_front(drv_pkt);

end
join_none
forever
begin
recv_port.get(recv_pkt);

```

```

this.no_rcv_pkt++;
exp_pkt = exp_que.pop_back();

$swrite(msg,"Recieved packet no %d\n",no_rcv_pkt);
avm_report_message("sb",msg);
if(rcv_pkt.comp(rcv_pkt,exp_pkt))
avm_report_message("sb"," Packet matched ");
else
avm_report_error("sb"," Packet mismatch ");

end
endtask

endclass

```

`endif

## VMM Ethernet

This testbench is developed in VMM (Systemverilog) for the Ethernet core available from opencores.org. My intension here is to explore the VMM methodology but not to verify the Ethernet core, as a result there are many bugs in the environment. I dont remember the versions of VMM but I developed these in the third quarter of 2007. To simulate this testbench some dependencies on libraries has to be removed from RTL files. It takes bit time for these changes in RTL.

### **Feauters:**

- Full support of automatic random, constrained random, and directed testcase creation.
- Supports injection of random errored packets.
- Supports 1G Full duplex modeled both in RX and TX paths.
- Protocol Checker/Monitor for self checking.
- Built in function coverage support for packets.
- Developed in Systemverilog using Synopsys VMM base classes.

**NOTE:** All trademarks are the property of their respective owners.

[Download vmm.tar](#)

[Browse the code in vmm\\_<eth>.tar](#)

## BLOCK DIAGRAM OF ETHERNET VERIFICATION ENVIRONMENT

