FINE GRAIN PROCESS CONTROL

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
shortint	2	Signed	16	SystemVerilog
int	2	Signed	32	SystemVerilog
longint	2	Signed	64	SystemVerilog
byte	2	Signed	8	SystemVerilog
bit	2	Unsigned		SystemVerilog
logic	4	Unsigned		SystemVerilog
reg	4	Unsigned		Verilog
integer	4	Signed	>= 32	Verilog
real				Verilog
shortreal				SystemVerilog
realtime				Verilog
time	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'hffffffff;
```

Systemverilog Adds the ability to specify unsized literal single bit values with a preceding (').'0, '1, 'X, '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 Pattren 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 allos the user to define datatypes. There are different ways to define user defined datatypes. They are
1. Class.
2. Enumarations.
3. Struct.
4. Union.
5. Typedef.

ENUMARATIONS

You'll sometimes be faced with the need for variables that have a limited set of possible values that can be usally referred to by name. For example, the state variable like IDLE,READY,BUZY etx 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 then , each one would have a unique value. By defaule 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;
```

Enumarated 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 UNIOUNS

Structure:

The disadvantage of arrays is that all the elements stored in then 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.

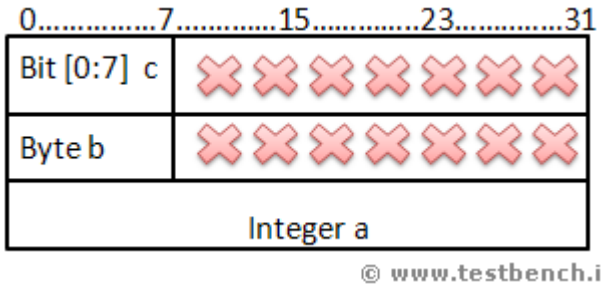
```
my_data_struct = `{a:1234,default:8'h20};
```

Union

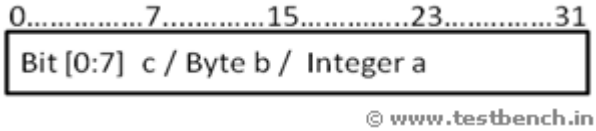
Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area. A union allows us to treat the same space in memory as a number of different variables. That is a Union offers a way for a section of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



Packed Structures:

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

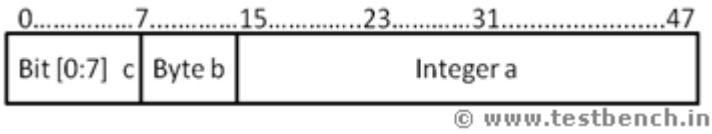
For example

```
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".



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
```

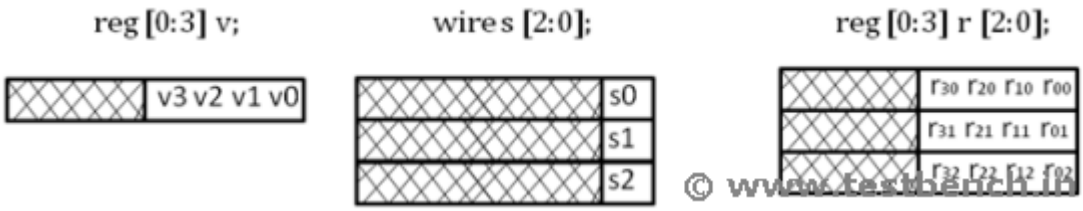
ARRAYS

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.

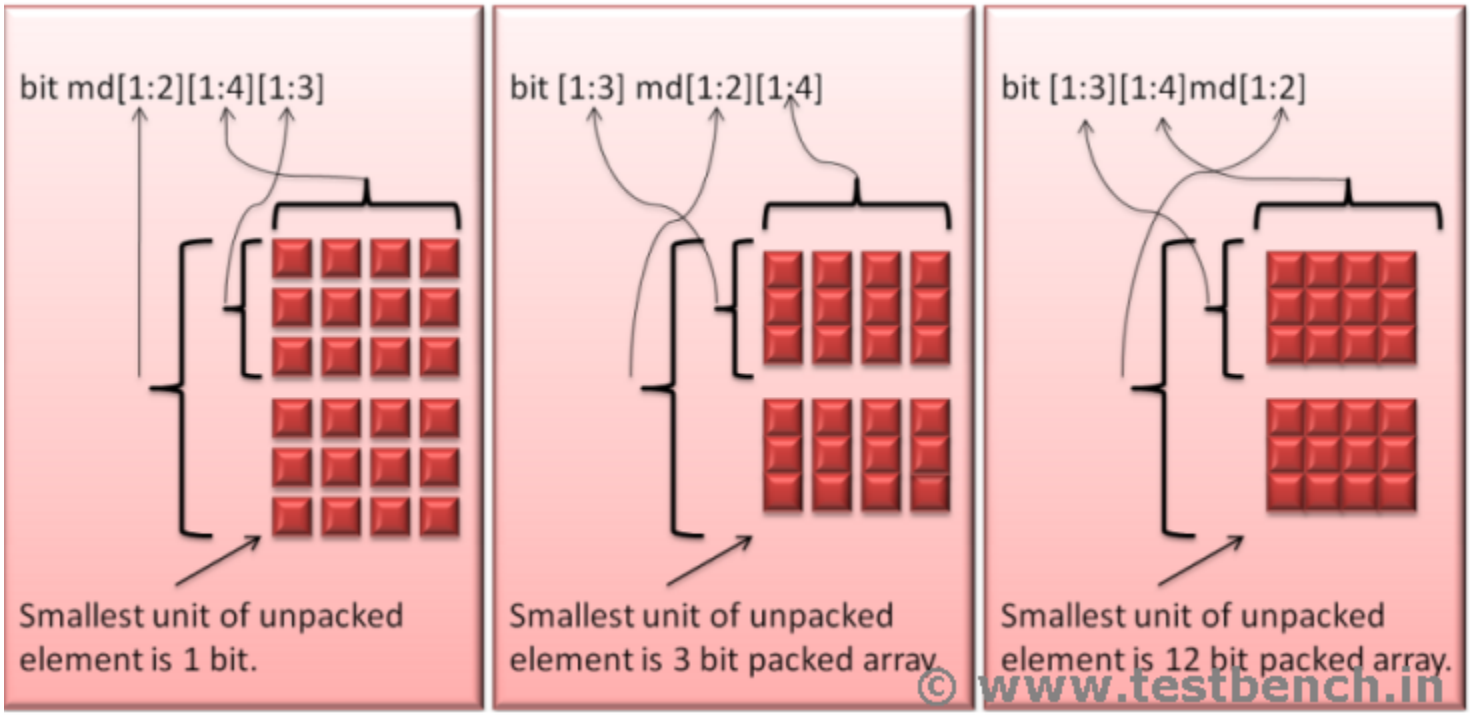
Fixed Arrays:

"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

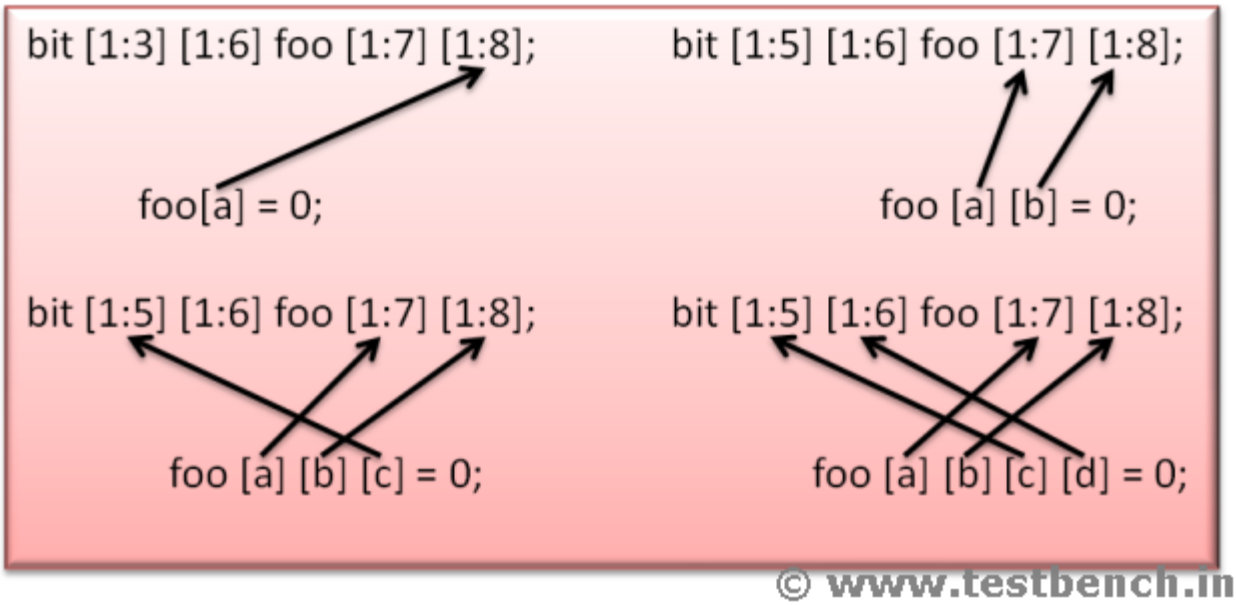
The following operations can be performed on all arrays, packed or unpacked:

```
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.



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.

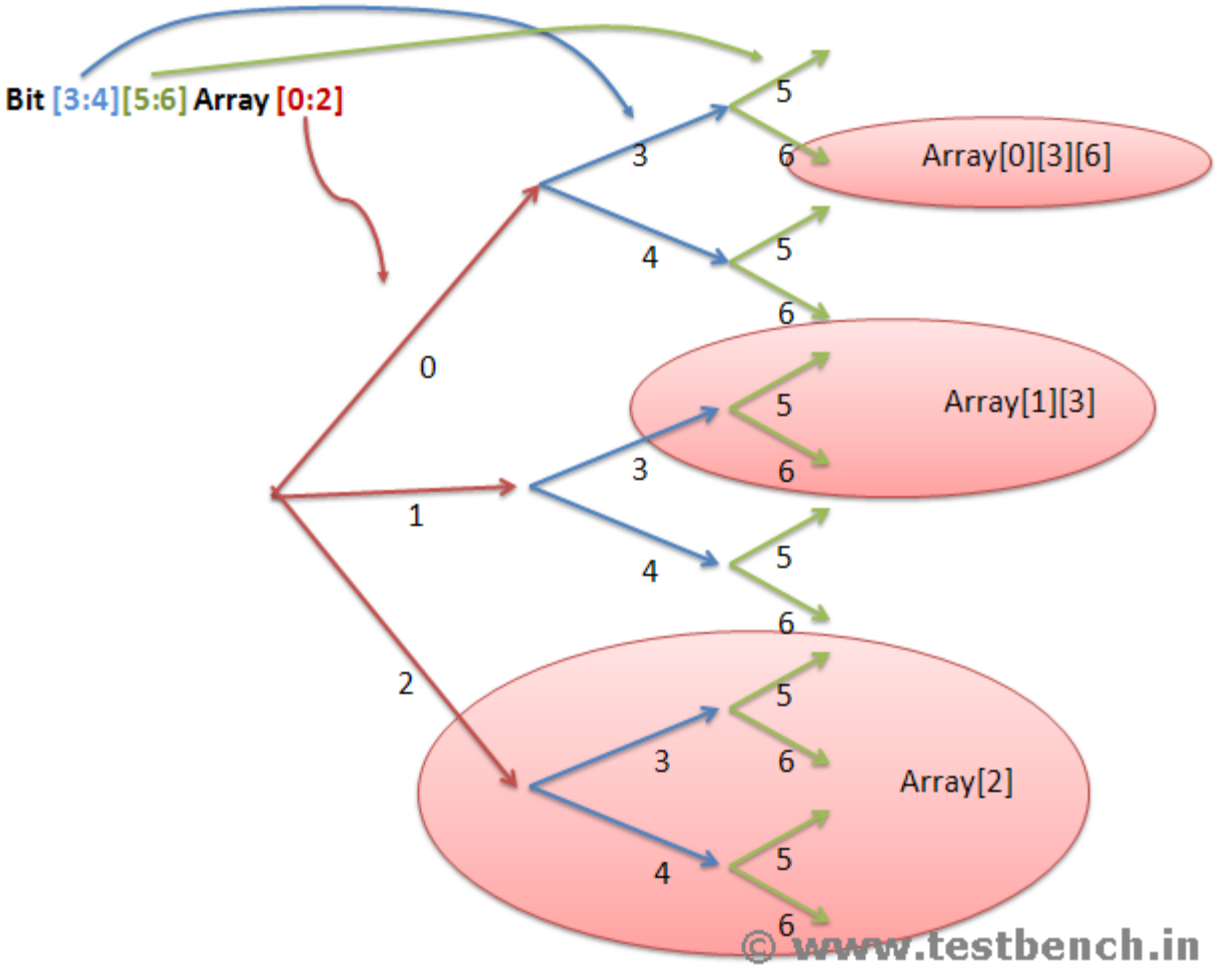
```
module array();
bit [1:5][10:16] foo1 [21:27][31:38],foo2 [31:27][33:38];
initial
begin
$display(" dimensions of foo1 is %d foo2 is %d",$dimensions(foo1),$dimensions(foo2) );
$display(" reading with out of bound resulted %d",foo1[100][100][100][100]);
$display(" reading with index x resulted %d",foo1[33][1'bx]);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

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

(S)$right

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

(S)$low
$low shall return the minimum of $left and $right of the dimension.

(S)$high
$high shall return the maximum of $left and $right of the dimension.

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

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

(S)$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) :

(S)find()
find() returns all the elements satisfying the given expression

(S)find_index()
find_index() returns the indexes of all the elements satisfying the given expression

(S)find_first()
find_first() returns the first element satisfying the given expression

(S)find_first_index()
find_first_index() returns the index of the first element satisfying the given expression

(S)find_last()
find_last() returns the last element satisfying the given expression

(S)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.

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

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

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

(S)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"};
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:

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

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

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

(S)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:

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

(S)product()
product() returns the product of all the array elements

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

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

(S)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 the 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 require 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 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 Dynmic 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~Rs 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:

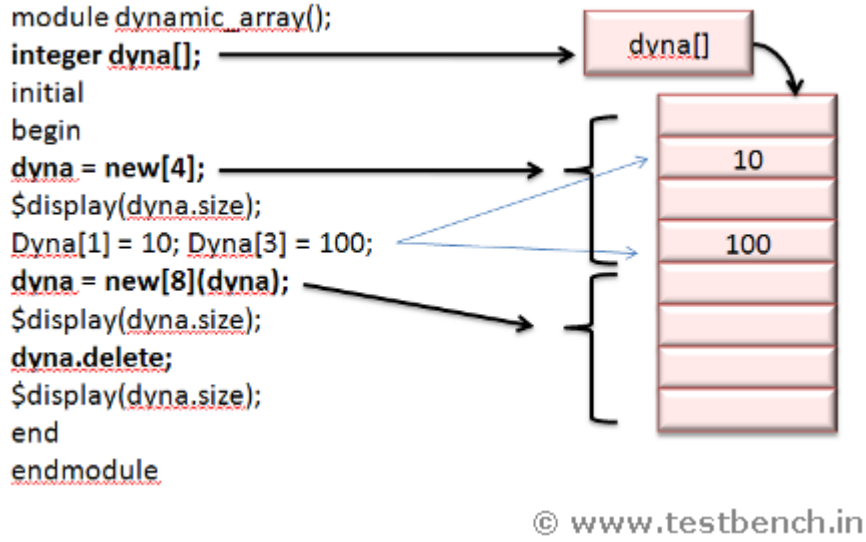
Using new[] constructor and its argument, we can increase the array size without losing the data content.

```
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.
```




```
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'hffff ] = 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 indexs
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.

Associative Array

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.

Dynamic Array

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.

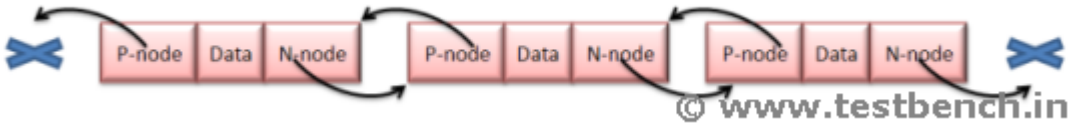
Queues

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
Fixed	Ok	Good	Ok
Associative	Good	Ok	Ok
Dynamic	Good	Good	Ok
Queue	Good	Good	Very Good

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.



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.sto 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 assiginment 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 terns 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

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;

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.
Â´{} 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(" -(nagetion) 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

```
-(nagetion) 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


```
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_pos_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**op2_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_po
s**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_neg_even,op1_
pos**op2_neg_even);
end
endprogram
```

RESULT

```
| -10 -1 0 1 10
```

```
---|-------------------------------------------------------
9| 3294967296 4294967295 0 1 1000000000
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

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 %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("-------------------------");
$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

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)).

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 %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 (" %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
~	Bitwise negation (unary)
&	Bitwise and (binary)
&~	Bitwise nand (binary)
|	Bitwise or (binary)
|~	Bitwise nor (binary)
^	Bitwise exclusive or (binary
^~ , ~^	Bitwise exclusive nor (binary)

In Systemverilog, bitwise exclusive nor has two notations (~^ and ^~).

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

EXAMPLE : Reduction

```
program main ;
reg [3:0] a_1,a_0,a_01xz,a_1xz,a_0xz,a_0dd1,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_0dd1 = 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 %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 %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

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 reverseoperation, i.e., unpack a stream of bits intoone **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 thr 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 thr 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 trigging 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 ");
#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 in 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 an 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);
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 multidimentional 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 : foeach
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.

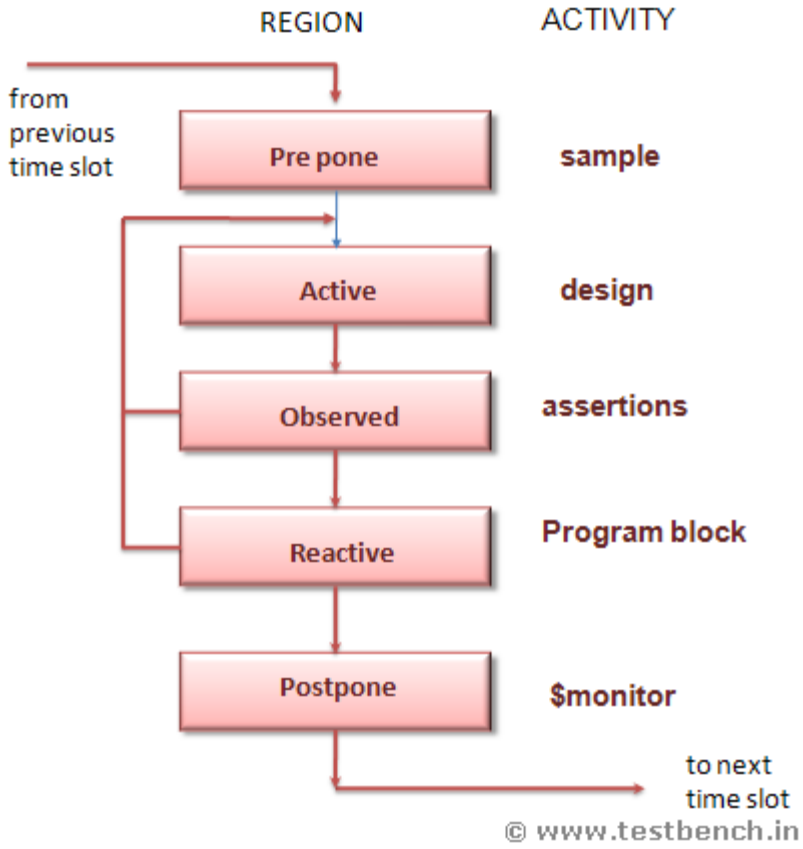
The program block serves these basic purposes:

-> 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 doesnot 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 testbenchs.

```
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

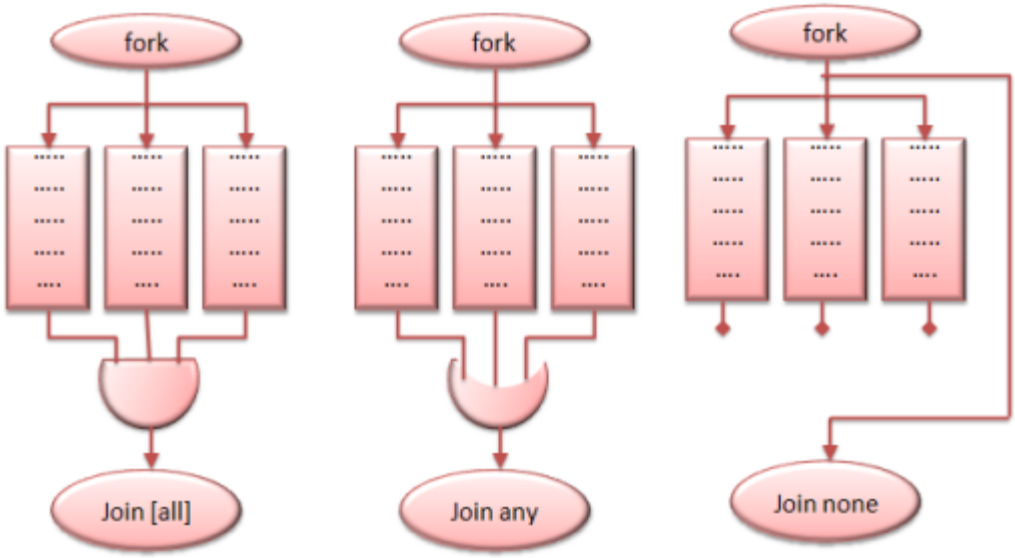
always_ff block to indicate sequential logic.

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 copyed 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 subrotunies are called,arguments those are omited,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" );
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
```

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 meain ;
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).

(S)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.

(S)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.

(S)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.

(S)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
```