Dynamic Array

- Dynamic arrays are unpacked arrays whose size can be set and changed during simulation time.
- new constructor is used to set or change size of Dynamic Array.
- size() method returns current size of array.
- delete() method is used to delete all elements of the array.

Dynamic Array

```
int dyn1 [ ];          //Defining Dynamic Array (empty subscript)
int dyn2 [4] [ ];

initial
begin
dyn1=new[10];                              //Allocate 10 elements
foreach (dyn1[ i ]) dyn1[ i ]=$random;    // Initializing Array
dyn1=new[20] (dyn1);                       // Resizing array and
                                  // Copying older values
dyn1=new[50];    // Resizing to 50 elements Old Values are lost
dyn1.delete;                               // Delete all elements
end
```

Dynamic Array

```
int dyn1 [ ]= ‘{5, 6, 7, 8} ;        //Alternative way to define size

initial
begin
  repeat (2)
   if (dyn1.size != 0)
     begin
     foreach(dyn1 [ i ] ) $display(“dyn1[%0d]=%0d”, i, dyn[ i ] );
     dyn1.delete;
     end
  else
     $display(“Array is empty”);
end
```

Queue

- A Queue is a variable size, ordered collection of homogenous elements.
- Queues support constant time access to all its elements.
- User can Add and Remove elements from anywhere in a queue.
- Queue is analogous to 1-D array that grows and shrinks automatically.
- 0 represents 1st element and $ represents last element.

Queue

Declaration:

```
int q1 [ $ ];                 // Unbounded Queue
int q2 [ $ : 100 ];           // Bounded Queue max size is 101
```

Operators:

```
q [ a : b ];
```

0 < a < b returns queue with b - a + 1 elements.

a = b = n returns q[n]

a > b returns empty queue

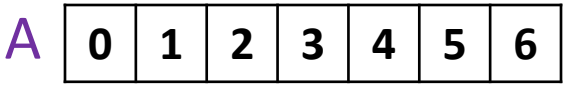
a or b is either x or z returns empty queue

a < 0 returns q [0: b]

b > $ returns q [a:$]

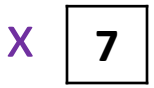
Queue Methods

```
int A [$] = '{ 0, 1, 2, 3, 4, 5, 6 };
int x, y, z;
```



- size() method returns number of elements in a queue.

```
x=A.size();
```

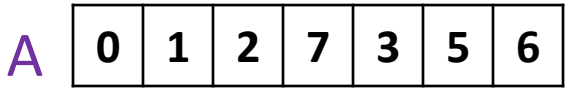


- insert(index, item) method is used to insert item at a given index.

```
A.insert(3, 7);
```

- delete(index) method is used to delete a queue if index is not specified else it is used to delete item at given index.

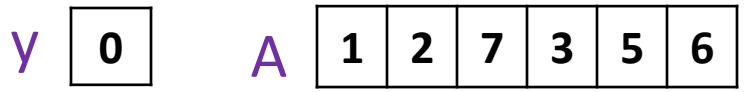
```
A.delete(5);
```



Queue Methods

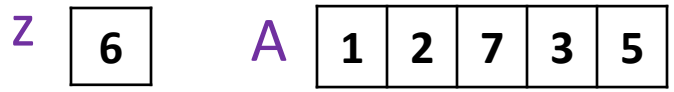
- pop_front() method removes and returns 1st element of the queue.

```
y=A.pop_front();
```



- pop_back() method removes and returns last element of the queue.

```
z=A.pop_back();
```

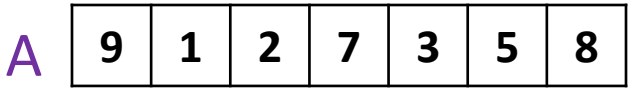


- push_front(item) method inserts item at the front of the queue.

```
A.push_front(9);
```

- push_back(item) method inserts item at the back of the queue.

```
A.push_back(8);
```



Queue

```
int q [$] = '{ 5, 7, 9, 11, 2};

q = { q, 6 };                                   // q.push_back(6)
q = { 3, q };                                   // q.push_front(3)
q = q [1:$];                                    // void'(q.pop_front())
                                                // or q.delete(0)
q = q[0:$-1];                                   // void'(q.pop_back())
                                                // or q.delete(q.size-1)
q = { q[0:3], 9, q[4:$] };                      // q.insert(4, 9)
q = {};                                         // q.delete()
q = q[2:$];              // a new queue lacking the first two items
q = q[1:$-1];        // a new queue lacking the first and last items
```

Array Locator Methods

- Array locator methods works on unpacked arrays and returns queue.
- with clause is mandatory for the following locator methods:
- find() returns all the elements satisfying the given expression.
- find_index() returns the indices of all the elements satisfying the given expression.
- find_first() returns the first element satisfying the given expression.

Array Locator Methods

- find_first_index() returns the index of the first element satisfying the given expression.
- find_last() returns the last element satisfying the given expression.
- find_last_index() returns the index of the last element satisfying the given expression.

Array Locator Methods

- with clause is not mandatory for the following locator methods:
- min() returns the element with the minimum value or whose expression evaluates to a minimum.
- max() returns the element with the maximum value or whose expression evaluates to a maximum.
- unique() returns all elements with unique values or whose expression evaluates to a unique value.
- unique_index() returns the indices of all elements with unique values or whose expression evaluates.

Array Locator Methods

```
int a [6] = ‘{9, 1, 8, 3, 4, 4};
int b [$], c [$] = ‘{1, 3, 5, 7};

b = c.min;                                  // {1}
b = c.max;                                  // {7}
b = a.unique;                               // {1, 3, 4, 8, 9}
b = a.find with (item > 3);                 // {9, 8, 4, 4}
b = a.find_index with (item > 3);           // {0, 2, 4, 5}
b = a.find_first with (item > 3);           // {9}
b = a.find_first_index with (item==8);      // {2}
b = a.find_last with (item==4);             // {4}
b = a.find_last_index with (item==4);       // {5}
```

Array Ordering Methods

- reverse() reverses the order of elements in an array.
- sort() sort array in ascending order with optional with clause.
- rsort() sort array in descending order with optional with clause.
- shuffle() randomizes the order of elements in an array.

```
int A [7] = '{ 5, 3, 1, 9, 8, 2, 7};
```

A | 5 | 3 | 1 | 9 | 8 | 2 | 7 |

```
A.reverse();
```

A | 7 | 2 | 8 | 9 | 1 | 3 | 5 |

```
A.sort();
```

A | 1 | 2 | 3 | 5 | 7 | 8 | 9 |

```
A.rsort();
```

A | 9 | 8 | 7 | 5 | 3 | 2 | 1 |

Array Reduction Methods

- sum() returns sum of all elements in an array or specific elements if with clause is present.
- product() returns product of all elements in an array or specific elements if with clause is present.
- and() returns bitwise and of all array elements or specific elements if with clause is present.
- or() returns bitwise or of all array elements or specific elements if with clause is present.
- xor() returns bitwise xor of all array elements or specific elements if with clause is present.

Associative Array

- In case size of data is not known or data space is sparse, Associative array is a better option.
- System Verilog allocates memory for an associative element when they are assigned.
- Index of associative can be of any type.
- If index is specified as * , then the array can be indexed by any integral expression of arbitrary size.
- real and shortreal are illegal index type.

Associative Array

```
int array1 [ * ];
int array2 [ int ];
//Array can be indexed by any integral expression.

int array3 [ string ];
//Indices can be strings or string literals of any length.

class xyz; …
int array4 [ xyz ];
//Indices can be objects of xyz.
```

Associative Array

```
int xyz [ * ];
```


```
xyz[0]=5;          //Memory allocated during assignment
xyz[1]=7;
xyz[2]=2;
xyz[3]=1;
xyz[7]=3;
xyz[10]=9;
```

Associative Array Methods

- num() and size() method returns number of elements in associative array.
- delete(index) deletes element at given index if index is specified else deletes entire array.
- exists(index) checks whether an element exists at the specified index.
- first(index) method assigns to the given index variable the value of the first (smallest) index. first (smallest) index. It returns 0 if the array is empty; otherwise, it returns 1.

Associative Array Methods

- last(index) 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.
- next(index) method finds the smallest index whose value is greater than the given index argument. Returns 1 if new index is different as old index else 0.
- prev(index) function finds the largest index whose value is smaller than the given index argument. Returns 1 if new index is different as old index else 0.

Associative Array Methods

```
int a [string]= ‘{“Jan”: 1, “Feb”: 2, “Mar”: 3, “April”: 4, “May”: 5};
string index;

initial
begin
a.first(index);                    //index=Jan
$display(a[index]);
while(a.next(index))               //Go through all index
$display(a[index]);
end
```

User Defined

- System Verilog allows user to define new data types using typedef keyword.

```
typedef byte unsigned uint8;          //Defining uint8
typedef bit [15:0] word;              //Defining word

uint8  a, b;
word c, d;

a=8'd10;
c=16'd25;
```

Structures

- Structure and Unions are used to group non-homogenous data types.
- By default structure are unpacked.
- Unpacked structure can contain any data type.

Declaration :

```
struct { bit [7:0] opcode;bit [15:0] addr; } IR;

struct {bit [7:0] r, g, b;} pixel;

struct {int a, b; real b;} mix;
```

Structures

Initializing :

```
IR='{opcode : 7'd8, addr : 15'd1};
pixel='{ 128, 255, 100};
pixel='{ r :128, g : 255, b :100};
pixel='{ int :0};
mix='{ 3, 5, 5.6};
mix='{ int : 1, real : 1.0};
mix='{ default : 0};
```

Accessing :

```
int x;
bit [7:0] y;
pixel.r=200;
mix.a=3;
mix.c=4.5;
x=mix.b;
y=pixel.g;
```

Packed Structures

- Packed Structure is made up of bit fields which are packed together in memory without gaps.
- A packed structure can be used as a whole to perform arithmetic and logical operations.
- First member of packed array occupies MSB and subsequent members follow decreasing significance.
- Structures can be packed by writing packed keyword which can be followed by signed or unsigned keyword.

Packed Structures

Example :

```
typedef struct packed signed { shortint a;   //16-bits [31:16]
                               byte b;       //8-bits   [15:8]
                               bit [7:0] c;  //8-bits    [7:0]
                              } exam_st;
exam_st pack1;

  bit [7:0] a, b, c;
  pack1='{a: '1, b: -10, c: 8'b1001_0101};
  a=pack1.b;
  b=pack1.c;
  c=pack1[9:2];
```

Packed Structures

- Only packed data type and integer data types are allowed inside packed structures

```
struct packed                    // default unsigned
{  bit [3:0] a;
   bit [7:0] b;
   bit [15:0] c [7:0] ; } pack2;
```

Compilation Error packed structure cannot have unpacked element

Packed vs Unpacked Structures

```
struct {  bit [7:0] a;
          bit [15:0] b;
          int c;
        } str1;
```

31:24	23:16	15:8	7:0
Unused			a
Unused		b	
c			

```
struct packed {  bit [7:0] a;
                 bit [15:0] b;
                 int c;
               } str2;
```

55:48	47:32	31:0
a	b	c

Unions

- Union represents a single piece of storage element that can be accessed by any of its member.

- Only one data types in union can be used at a time.

Example :

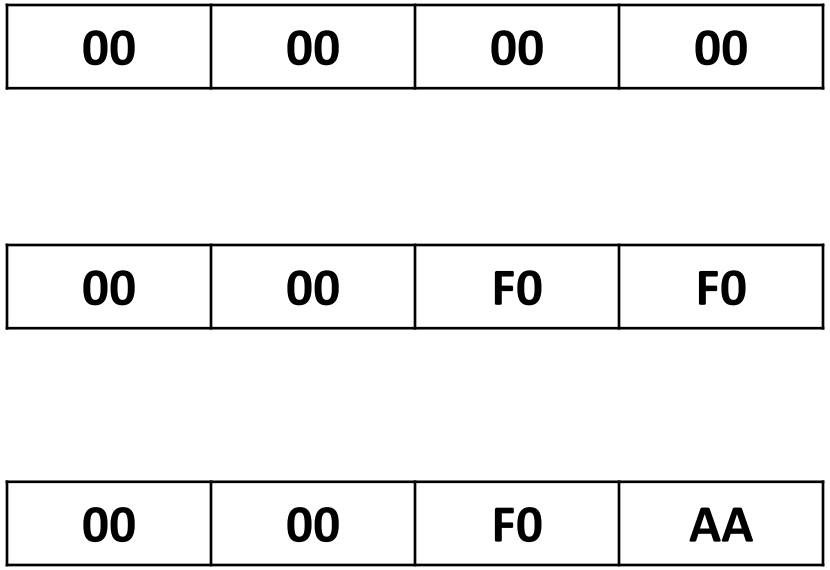
```
union
{  real a;
   int b;
   bit [7:0] c;  } exam1;
```

```
union packed
      {  real a;
         int b;
         bit [7:0] c;  } exam2;
```

Unions

Example :

```
typedef union
{  shortint a;
   int b;
   bit [7:0] c;  } my_un;
   my_un un1;
   un1.a=16'hf0f0;
   $displayh(un1.b);
   un1.c=8'b1010_1010;
   $displayh(un1.b);
```



Structures vs Unions

Structure	Union
Memory is allocated to each and every element.	Common memory is allocated for all the members.
Size of structure is sum of size of each member or more.	Size of union is equal to size of largest member
First member is at offset 0.	All member have 0 offset.
Modifying value of one member has no effect on other members	Modifying value of one member modifies value of all members

String

- System Verilog string type is used to store variable length strings.
- Each character of string is of type byte.
- There is no null character at the end of string.
- String uses dynamic memory allocation, so size of string is no longer a concern.

Example :

```
string s="hello";
```

String Operators

- str1 == str2 checks whether strings are equal or not.
- str1 != str2 checks for inequality of strings.
- Comparison using lexicographical ordering of strings.
  - str1 < str2
  - str1 <= str2
  - str1 > str2
  - str1 >= str2
- {str1, str2, str3, .. , strn} concatenation of strings.

String Operators

Example :

```
string s1="hello", s2="Hello", s3="xyz";
initial
begin
if(s1 != s2)
$display("strings are different");
if(s1 > s3)
$display("s1 is more than s3");
else
$display("s3 is more than s1");
$display({s1, s2, s3});
end
```

String Methods

- len() method returns length of a string.
- putc(position, character) method replaces character at given position by character passed as an argument.
- getc(position) method returns ASCII value of character at given position.
- toupper() method returns a string with all characters in uppercase.

String Methods

- tolower() method returns a string with all characters in lowercase.
- compare(string) compares given string with string passed as an argument.
- icompare(string) same as above but comparison is case insensitive.
- substr(i, j) returns a string formed between characters at position i and j.

String Methods

Example :

```
string s1, s2;
initial begin
s1 = "SystemVerilog";
$display(s1.getc(0));                    //Display: 83 ('S')
$display(s1.toupper());                  // Display: SYSTEMVERILOG

s1 = {s1, "3.1b"};                       // "SystemVerilog3.1b"

s1.putc(s1.len()-1, "a");                // change b-> a

$display(s1.substr(2, 5));               // Display: stem

s2=$psprintf("%s %0d", s1, 5);
$display(s2);                            // Display: SystemVerilog3.1a 5
end
```

Enumerated Type

- An enumeration creates a strong variable type that is limited to a set of specified names.

  Example :
  ```
  enum { RED, GREEN, BLUE } color;
  typedef enum { FETCH, DECODE, EXECUTE } operation_e;
  ```

- enum are stored as int unless specified.

  ```
  typedef enum bit [2:0] { RED, GREEN, BLUE } color_e;
  ```

- First member in enum gets value 0, second value 1 and so on.

- User can give different values to member if required.

Enumerated Type

Example :

```
enum { RED, GREEN, BLUE } color;
//RED=0, GREEN=1, BLUE=2

enum { GOLD, SILVER=3, BRONZE} medals;
//GOLD=0, SILVER=3, BRONZE=4

enum {A=1, B=3, C, D=4} alphabet;
//Compilation error C and D have same value

enum logic [1:0] {A=0; B='Z, C=1, D} exam;
//A=00, B=ZZ, C=01, D=10 Default value of exam is X
```

Enumerated Type Methods

- first() method returns first member of enumeration.
- last() method returns last member of enumeration.
- next(N) method returns the Nth next member (default is 1) starting from current position.
- previous(N) method returns Nth previous member (default is 1) starting from current position.

Enumerated Type Methods

- Both next() and prev() wraps around to start and end of enumeration respectively.
- num() method returns number of elements in given enumeration.
- name() method returns the string representation of given enumeration value.

Enumerated Type Methods

Example

```
typedef enum { RED, BLUE, GREEN} color_e;
color_e mycolor;
Initial begin
mycolor = mycolor.first;
do
begin
$display("Color = %0d %0s", mycolor, mycolor.name);
mycolor = mycolor.next;
end

while (mycolor != mycolor.first);   // Done at wrap-around
end
```

Casting

- Casting is used convert data from one type to other.

- There are two ways to perform casting :
  - Static Casting: destination = return_type’ (source). This type of casting always succeeds at run time and does not give any error.
  - Dynamic Casting: using $cast system task or function.

Example :

```
int a;
initial a=int’(3.0 * 2.0);
```

Casting

- System Verilog 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.

  $cast used as a function

  ```
  if ($cast(destination, source))    //destination and source
                                     // should be singular
  ```

  $cast used as a task

  ```
  $cast(destination, source);
  ```

Casting

```
int a;
real b=3.0;

if($cast(a, b))   //Returns 1 if casting succeeds else 0
$display("casting success");

$cast(a, b);     //If casting fails run time error occurs
```

In both cases if casting fails then destination value remains unchanged.

Casting

```
typedef enum { red, green, blue, yellow, white, black } Colors;
Colors col;
int a, b;

initial begin
col=green;
//col=3;              Runtime error
a= blue * 2;
b= col + green;
end
```

Casting

```
typedef enum { red, green, blue, yellow, white, black } Colors;
Colors col;

initial begin
$cast( col, 2 + 3 );                    //col=black

if ( ! $cast( col, 2 + 8 ) )            //10: invalid cast
$display( "Error in cast" );

col = Colors'(2 + 1);                   //col=yellow
col = Colors'(4 + 3);                   //value is empty
end
```