

Arrays

Agenda

- Differences b/w Verilog and SV arrays
- Introduction
- Packed Vs Unpacked Arrays
- Initialization, Assignment of Arrays
- Array manipulator methods
- Dynamic Arrays
- Associative Arrays
- Queue Arrays
- Comparison of Dynamic, Associative, Queuing Arrays.

Differences b/w Verilog arrays and SV arrays

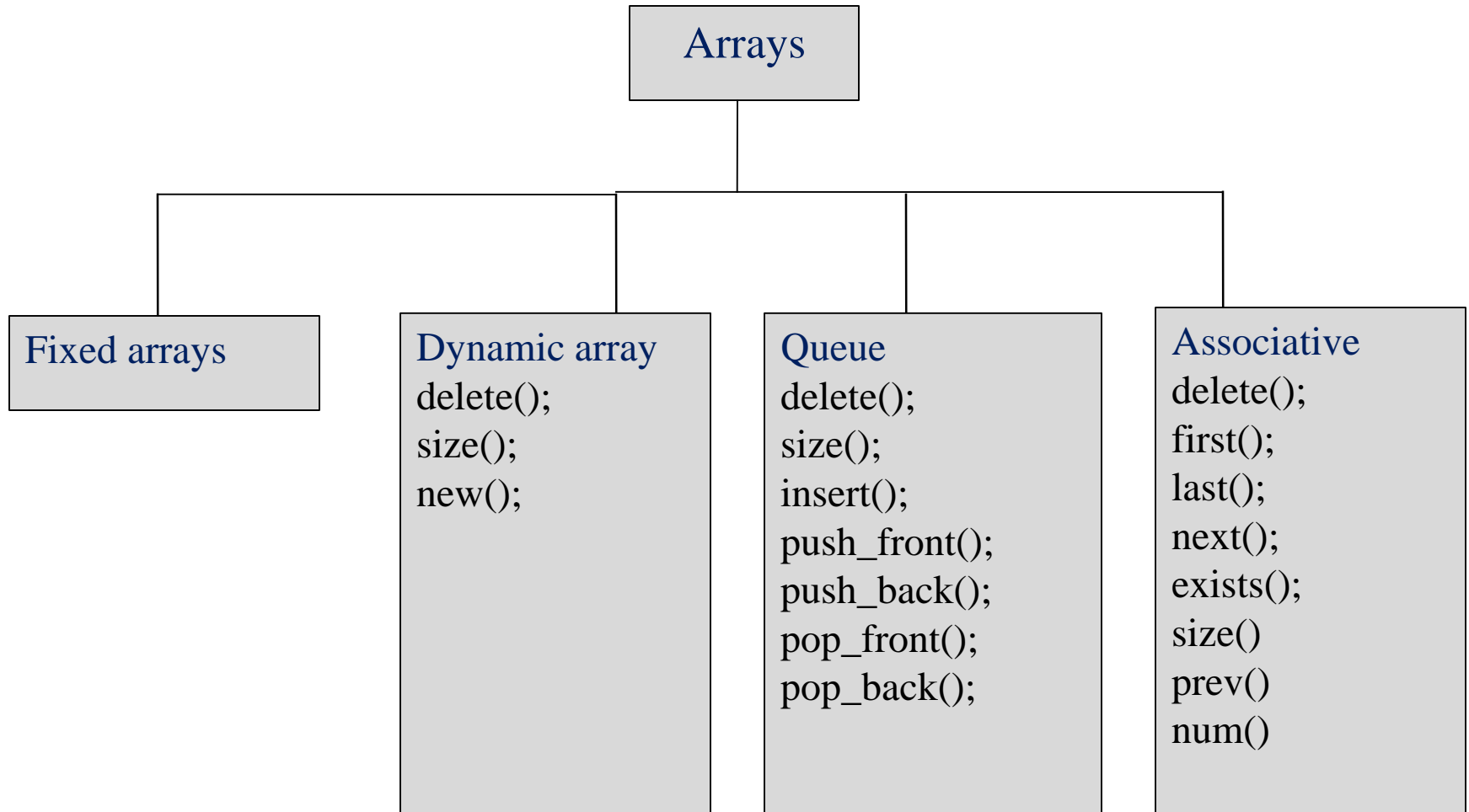
Verilog

- Verilog does not have user defined types
- Fixed Arrays
- One element is initialized at each time
- One element is copied at each time
- Plain, simple, but quite limited
- No dynamic allocation

System Verilog

- User defined data types
- Unpacked, Packed arrays
- More than one element is initialized
- More than one element is copied to another array of same data type
- more flexible with more features
- Dynamic allocation

Introduction to Arrays



Contd..

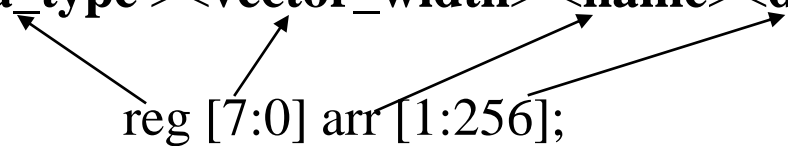
Array is collection of variables, all of same type, and accessed using the same name plus one or more indices

Arrays are indexed from left bound to right bound

Syntax: **<data_type>** **<vector_width>** **<name>** **<dimension>**

Example:

reg [7:0] arr [1:256];
wire [7:0] arr [256];



Examples:

```
wire num [0:1023];
```

// a 1-dimensional unpacked array of 1024 1-bit nets

```
reg [7:0] LUT [0:255];
```

// a 1-dimensional unpacked array of 256 locations of data width 8

```
reg [7:0] LUT [3:0][0:255];
```

// 4 X 256 locations, each of data width 8

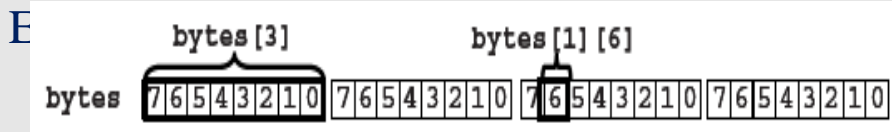
Packed Vs unpacked arrays

Packed Arrays

System Verilog uses the term “*packed array*” to refer to the dimensions declared before the object name (Verilog-2001 refers to as the **vector width**).

```
bit [7:0] c1;           // c1 is a 8bit variable
```

```
bit [3:0] [7:0] bytes;  // packed array- 4  
elements ,each of width 8bit
```

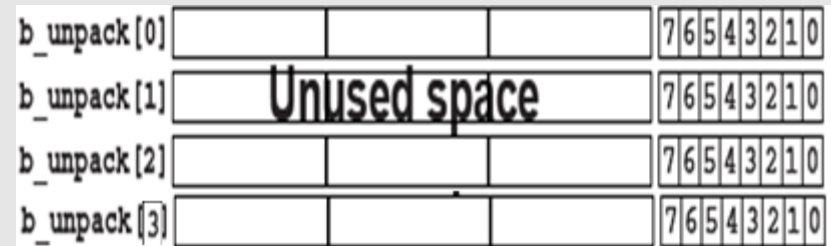


Unpacked arrays

The term “*unpacked array*” is used to refer to the dimensions declared after the object name.

```
bit [7:0] b_unpack [3:0]; // unpacked array
```

Example:



Contd..

- Packed arrays can only be made of the single bit types (**bit**, **logic**, **reg**, **wire** and the other **net** types) .
- It can access whole array or slice as a vector or only a bit.
- Integer types with predefined widths cannot have packed array declared. These types are: **byte**, **shortint**, **int**, **longint** and **integer**.
- An integer type with a predefined width can be treated as a single dimension packed array. These types shall be numbered down to 0.

```
byte c2;           // same as bit [7:0] c2;  
integer i1;        // same as logic signed [31:0] i1;
```

Using packed and unpacked arrays

Packed arrays to model:

- Vectors where it is useful to access sub-fields of the vector

Example:

```
logic [39:0][15:0] packet; // 40 location each of 16-bit
```

```
packet = input_stream; // assign to all words
```

```
data = packet[24]; // select the 16-bit of 24th location
```

```
tag = packet[3][7:0]; // select the 8-bit of 3rd location
```

Unpacked arrays to model:

- Arrays where typically one element at a time is accessed, such as with RAMs and ROM

Example:

```
module ROM (...);
```

```
byte mem [0:4095]; //unpacked array of bytes
```

```
assign data = select? mem[address]: 'z;
```


Initialization Of Arrays

Packed Array Initialization

```
logic [3:0][7:0] a = 32'h0;    // vector assignment  
logic [3:0][7:0] b = {16'hz,16'h0}; // concatenate operator  
logic [3:0][7:0] c = {16{2'b01}}; // replicate operator
```

Unpacked array initialization

```
int d [0:1][0:3] = '{ ' {7,3,0,5}, ' {2,0,1,6} };  
    // d[0][0] = 7  
    // d[0][1] = 3  
    // d[0][2] = 0  
    // d[0][3] = 5  
    // d[1][0] = 2  
    // d[1][1] = 0  
    // d[1][2] = 1  
    // d[1][3] = 6
```

Assigning Values to Unpacked Arrays

Specifying a default value to the unpacked array

- System Verilog provides a mechanism to initialize all the elements of an unpacked array or a slice of an unpacked array, by *specifying a default value*.
- The value assigned to the array must be compatible with the type of the array. A value is compatible if it can be **cast** to that type i.e., assigning the packed to unpacked.

```
integer a1 [0:7][0:1023] = '{default : 8'h55};
```

- We can also assign values to individual elements or slice or part of the array.

Example:

```
byte a [0:3][0:3];
```

```
a[1][0] = 8'h5; // assign to one element
```

```
a = '{ '{0,1,2,3}, '{4,5,6,7}, '{7,6,5,4}, '{3,2,1,0}}; // assign a list of  
values to the full array
```

```
a[3] = '{ 'hF, 'hA, 'hC, 'hE}; // assign list of values to slice of the array
```

Example

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

Un-Packed array

```
bit [7:0] array4[2:0];
```

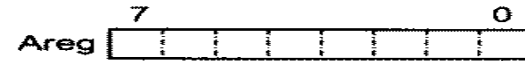
[illegible]

Packed array

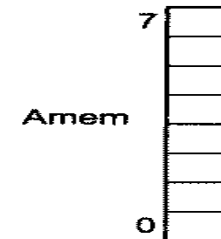
```
bit [2:0] [7:0] array5;
```

							7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
							array5[2]							array5[1]							array5[0]									

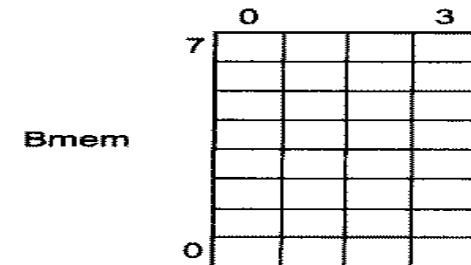
```
// An 8-bit vector
reg [7:0] Areg;
```



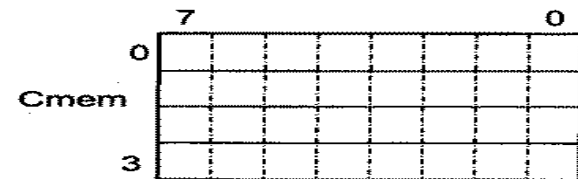
```
// A memory of 8 one-bit elements
reg Amem [7:0];
```



```
// A two-dimensional memory of one-bit elements
reg Bmem [7:0] [0:3];
```



```
// A memory of four 8-bit words
reg [7:0] Cmem[0:3];
```



```
// A two-dimensional memory of 3-bit elements
reg[2:0] Dmem [0:3] [0:4];
```

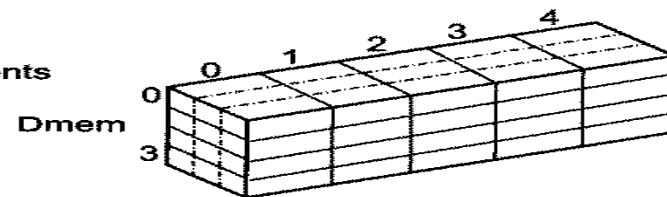
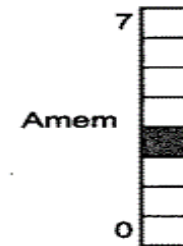


Figure 3.8 Array structures

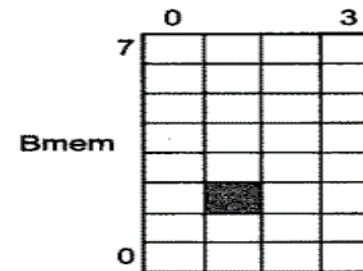
```
// declaration: reg [7:0] Areg;  
Areg [7:5]
```



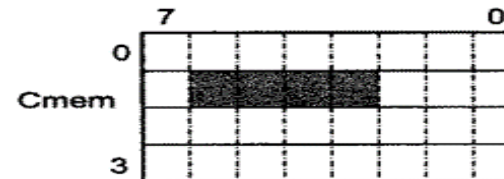
```
// declaration: reg Amem [7:0];  
Amem [3]
```



```
// declaration: reg Bmem [7:0][0:3]  
Bmem [2] [1]
```



```
// declaration: reg[7:0] Cmem [0:3]  
Cmem [1] [6 -: 4]
```



```
// declaration: reg [2:0] Dmem [0:3][0:4]  
Dmem [0] [2]
```

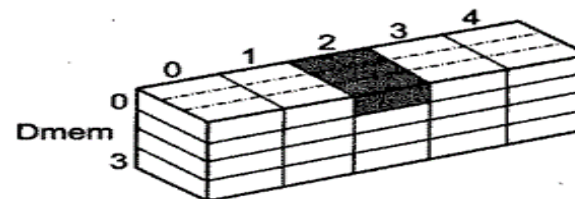


Figure 3.16 Array Addressing and Selection

Copying Packed Arrays

Packed arrays can be copied/assigned to another packed array of same element size

```
bit [1:0][15:0] a;    // 32 bit 2-state vector
```

```
logic [3:0][ 7:0] b;  // 32 bit 4-state vector
```

```
logic [15:0] c;       // 16 bit 4-state vector
```

```
logic [39:0] d;       // 40 bit 4-state vector
```

```
b = a;                // assign 32-bit array to 32-bit array
```

```
c = a;                // upper 16 bits will be truncated
```

```
d = a;                // upper 8 bits will be zero filled
```

Array manipulator method

```
graph TD; A[Array manipulator method] --> B[Array locator method]; A --> C[Array ordering method]; A --> D[Array reduction method];
```

Array locator method

find();
find_index();
find_first();
find_first_index();
find_last_index();
min();
max();
unique();
unique_index();

Array ordering method

reverse();
sort();
rsort();
shuffle();

Array reduction method

sum();
product();
and();
or();
xor();

Array Locator methods

- **find()** with **(filter_expr)**: return all elements satisfying the given expression
- **find_index()** with **(filter_expr)**: returns the indexes of all elements satisfying the given expression
- **find_first()** with **(filter_expr)**: returns the first element satisfying the given expression
- **find_first_index()** with **(filter_expr)**: returns the index of the first element satisfying the given expression
- **find_last()** with **(filter_expr)**: returns the last element satisfying the given expression
- **find_last_index()** with **(filter_expr)**: returns the index of the last element satisfying the given expression
- **min()**: returns the element with the minimum value
- **max()**: returns the element with the maximum value
- **unique()**: returns all the elements with the unique values
- **unique_index()**: returns all the indexes with the unique values

Contd..

Order methods:

- **reverse():** reverses all the elements of the array
- **sort():** sorts the unpacked array in the ascending order
- **rsort():** sorts the unpacked array in the descending order
- **shuffle():** randomize the order of the elements in array

Reduction methods:

- **sum():** returns sum of all the array elements
- **product():** returns product of all the array elements
- **and():** returns the bitwise AND(&) of all the array elements
- **or():** returns the bitwise OR(|) of all the array elements
- **xor():** return the logical XOR(^) of all the array elements

Dynamic Arrays

- A dynamic array is one dimension of an unpacked array whose size can be set or **changed at runtime**.
- Space doesn't exist until the array is explicitly created at runtime.

Syntax : `data_type array_name [];`

- Dynamic arrays support the same types as fixed-size arrays.

Example:

`bit [3:0] nibble[];` // Dynamic array of 4-bit vectors

`integer mem [];` // Dynamic array of integers

Dynamic Array Functions:

The `new[]` operator is used to set or change the size of the array.

The `size()` built-in method returns the current size of the array.

`function int size();`

The `delete()` built-in method clears all the elements yielding an empty array

`function void delete();`

Contd..

```
int my_array [];
```

Example:1

```
initial begin
    my_array = new[4]; //Allocated 4 elements
    my_array={ 1,2,3,4};
    $display("my_array=%p",my_array);
end
```

o/p 1:my_array=1,2,3,4

Example:2 To resize a dynamic array

```
initial begin
    my_array = new[5](my_array); //Resize the Array and Copy
    $display("my_array=%d",my_array.size());
end
```

o/p 2:my_array=5

Example:3

```
initial begin
    my_array.delete(); //elements of my_array will be deleted.
    $display("my_array=%p",my_array.size());
end
```

o/p 3:my_array=0

```

module dynamic_array;
    //dynamic array declaration
    bit [7:0] d_array1[];
    int      d_array2[];

    initial begin
        $display("Before Memory Allocation");
        $display("\tSize of d_array1 %0d",d_array1.size());
        $display("\tSize of d_array2 %0d",d_array2.size());

//memory allocation
        d_array1 = new[4];
        d_array2 = new[6];

        $display("After Memory Allocation");
        $display("\tSize of d_array1 %0d",d_array1.size());
        $display("\tSize of d_array2 %0d",d_array2.size());

```

```

//array initialization
d_array1 = {0,1,2,3};
foreach(d_array2[j])
    d_array2[j] = j;

$display("--- d_array1 Values are ---");
foreach(d_array1[i])
    $display("\t d_aaray1[%0d] = %0d",i, d_array1[i]);
$display("-----");

$display("--- d_array2 Values are ---");
foreach(d_array2[i]) $display("\td_aaray2[%0d] = %0d",i, d_array2[i]);
$display("-----");
end

endmodule

```

Result

Before Memory Allocation

Size of d_array1 0

Size of d_array2 0

After Memory Allocation

Size of d_array1 4

Size of d_array2 6

--- d_array1 Values are ---

d_aaray1[0] = 0

d_aaray1[1] = 1

d_aaray1[2] = 2

d_aaray1[3] = 3

--- d_array2
Values are ---

d_aaray2[0] = 0

d_aaray2[1] = 1

d_aaray2[2] = 2

d_aaray2[3] = 3

d_aaray2[4] = 4

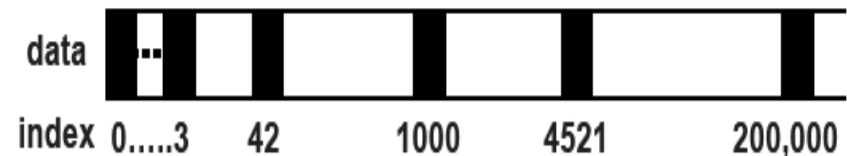
d_aaray2[5] = 5

Associate Arrays

- **Dynamic arrays** are useful for dealing with **contiguous** collections of variables whose number changes dynamically.
- When the **size of the collection** is unknown or the data space is sparse, an **associative array** is a better option.
- It doesn't have any storage allocated until it is used, and the index can be of any type.
- It implements a lookup table of the elements of its declared type.
- The data type to be used as an **index serves as the lookup key**

syntax: <data_type> <array_id> [<index_type>];

- It can address a very large address space.
- Below example, the associative array holds the values 0:3, 42, 1000, 4,521, and 200,000.
- The memory used to store these is far less than would be needed to store a fixed or dynamic array with 200,000 entries.



Contd..

integer i_array[*]; // associative array of integer Wild card index

bit [20:0] array_b[string]; // associative array of 21-bit vector, indexed by string

event ev_array[myClass]; // associative array of event indexed by myClass

- Array elements in associative arrays are allocated dynamically; an entry is created the first time it is written.
- The associative array maintains the entries that have been assigned values and their relative order according to the index data type.
- Associative array elements are unpacked, meaning that other than copying or comparing arrays, we have to select an individual element out of the array before using it.

Wildcard index type

```
int array_name [*];
```

Associative arrays that specify a wildcard index type have the following properties:

- The array may be indexed by any integral expression. Because the index expressions may be of different sizes, the same numerical value can have multiple representations, each of a different size.

SystemVerilog resolves this ambiguity by removing the leading zeros and computing the minimal length and using that representation for the value.

- Nonintegral index values are illegal and result in an error.
- A 4-state index value containing X or Z is invalid.
- Indexing expressions are self-determined and treated as unsigned.

String index

```
int array_name [ string ];
```

Associative arrays that specify a string index have the following properties:

- Indices can be strings or string literals of any length. Other types are illegal and shall result in a type check error.
- An empty string "" index is valid.

Class index

```
int array_name [ some_Class ];
```

Associative arrays that specify a class index have the following properties:

- Indices can be objects of that particular type or derived from that type. Any other type is illegal and shall result in a type check error.
- A null index is valid.

Integral index

Associative arrays that specify an index of integral data type shall have the following properties:

- The index expression shall be evaluated in terms of a cast to the index type, except that an implicit cast from a real or shortreal data type shall be illegal.
- A 4-state index expression containing X or Z is invalid.
- The ordering is signed or unsigned numerical, depending on the signedness of the index type.

Index Type

The array can be indexed by any **integral** data type with **wildcard** '*'.

Example: **int** array_name [*];

String index:

Example: **int** array_name [**string**];

Class index:

Example: **int** array_name [some_Class];

Integer (or int) index:

Example: **int** array_name [**integer**];

Signed packed array:

Example: **typedef bit signed** [4:1] Nibble;
int array_name [Nibble];

Associate Array Traversal Methods

num()

It returns the number of entries in the associative array. If the array is empty, it returns 0.

Syntax: function int **num()**;

Example: initial begin

```
int imem[*];  
$display( "%0d entries\n", imem.num );  
imem[2'b10] = 1;  
$display( "%0d entries\n", imem.num );  
imem[ 16'hffff ] = 2;  
$display( "%0d entries\n", imem.num );  
imem[4'b1000] = 3;  
$display( "%0d entries\n", imem.num );  
end
```

o/p:

0 entries

1 entries

2 entries

3 entries

delete()

If the *index* is specified, then the `delete()` method removes the entry at the specified index. If the entry to be deleted does not exist, the method issues no warning.

Syntax: `function void delete([input index]);`

If the '*index*' is not specified, then the '*delete()*' method removes all the elements in the array.

Example:

```
module tb;
initial begin
int map[ string ];
map[ "hello" ] = 1;
map[ "sad" ] = 2;
map[ "world" ] = 3;
    $display("map entries %d",map.num());
map.delete( "sad" ); // remove entry whose index is "sad" from "map"
    $display("map entries %d",map.num());
map.delete; // remove all entries from the associative array "map"
    $display("map entries %d",map.num());
end
endmodule
```

o/p:
map entries 3

map entries 2

map entries 0

Contd..

exists()

The **exists()** function checks if an element exists at the specified index within the given array. It returns 1 if the element exists, otherwise it returns 0.

Syntax: **function int exists(input index);**

Where *index* is an index of the appropriate type for the array.

Example: (from previous example)

initial begin

if (map. exists("hello"))

\$display(“value is available in map[%s]=%d ”, hello, map[hello]);

else

\$display(“value is unavailable in map[%s]=Null ”, hello);

end

o/p : value is available in map [hello]=1

Contd..

first()

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, and 1 otherwise.

Syntax: **function int** first(**ref** index);

Example:(from previous example)

```
initial begin
string s;
if ( map. first( s ) )
  $display( "First entry is : map[ %s ] = %0d\n", s, map[s] );
end
```

o/p: First entry is : map[hello]=1

Contd..

last()

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 and 1 otherwise.

Syntax: function int **last**(ref index);

Example:(from previous example)

```
initial begin
```

```
  string s;
```

```
  if ( map. last( s ) )
```

```
    $display( "Last entry is : map[ %s ] = %0d\n", s, map[s] );
```

```
end
```

o/p : Last entry is :map[word]=3

Contd..

next()

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, index is unchanged, and the function returns 0.

Syntax: function int **next**(ref index);

Example: (from previous example)

```
initial begin
string s;
if ( map. first( s ) )
do
$display( "%s : %d\n", s, map[ s ] );
while ( map. next( s ) );
end
```

o/p:

hello :1

Sad :2

World :3

Contd..

prev()

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.

Syntax: **function** `int prev(ref index);`

Example:(from previous example)

```
initial begin  
  string s;  
  if ( map. last( s ) )  
    do  
      $display( "%s : %d\n", s, map[ s ] );  
      while ( map. prev( s ) );  
    end
```

o/p:

world :3

sad :2

hello:1


```

module associative_array;
  //array declaration
  int a_array[*];
  int index;

  initial begin
    //allocating array and assigning value to it
    repeat(3) begin
      a_array[index] = index*2;
      index=index+4;
    end

    //num() –Associative array method
    $display("\tNumber of entries in a_array is %0d",a_array.num());
    $display("--- Associative array a_array entries and Values are ---");
    foreach(a_array[i])
      $display("\ta_array[%0d] \t = %0d",i,a_array[i]);
    $display("-----");
  //first()-Associative array method
  a_array.first(index);
  $display("\tFirst entry is \t a_array[%0d] = %0d",index,a_array[index]);

  //last()-Associative array method
  a_array.last(index);
  $display("\tLast entry is \t a_array[%0d] = %0d",index,a_array[index]);

```

//exists()-Associative array method

```
if(a_array.exists(8))  
    $display("Index 8 exists in a_array");  
else  
    $display("Index 8 doesnt exists in a_array");
```

//last()-Associative array method

```
a_array.last(index);  
$display("Last entry is a_array[%0d] = %0d",index,a_array[index]);
```

//prev()-Associative array method

```
a_array.prev(index);  
$display("entry is a_array[%0d] = %0d",index,a_array[index]);
```

//next()-Associative array method

```
a_array.next(index);  
$display("entry is a_array[%0d] = %0d",index,a_array[index]);  
end  
endmodule
```

Number of entries in a_array is 3
--- Associative array a_array entries and Values are ---

a_array[0] = 0
a_array[4] = 8
a_array[8] = 16

First entry is a_array[0] = 0
Last entry is a_array[8] = 16
Index 8 exists in a_array
Last entry is a_array[8] = 16
entry is a_array[4] = 8
entry is a_array[8] = 16

```

module tb;
  bit[3:0] Ass_array[string];
  string Ass_array_index;
  initial
  begin
    Ass_array = {"SIMPLE":5,"HUMBLE":10,"RESPECTIVE":20,"KINDNESS":30,"PUNCTUALITY":12};

    $display(Ass_array);
    //first
    $display("searching for first element",Ass_array.first(Ass_array_index));
    $display("Ass_array_index=%p",Ass_array_index);
    //next
    $display("searching for next element",Ass_array.next(Ass_array_index));
    $display("Ass_array_index=%p",Ass_array_index);
    //last
    $display("searching for last element",Ass_array.last(Ass_array_index));
    $display("Ass_array_index=%p",Ass_array_index);
    //prev
    $display("searching for prev element",Ass_array.prev(Ass_array_index));
    $display("Ass_array_index=%p",Ass_array_index);
  end
endmodule

```

Associative with string index example

```

{"HUMBLE":'ha, "KINDNESS":'he, "PUNCTUALITY":'hc, "RESPECTIVE":'h4, "SIMPLE":'h5}
searching for first element 1
Ass_array_index="HUMBLE"
searching for next element 1
Ass_array_index="KINDNESS"
searching for last element 1
Ass_array_index="SIMPLE:"
searching for prev element 1
Ass_array_index="RESPECTIVE"

```

```
module tb;
  bit[3:0] Ass_array[int];
  int Ass_array_index;
```

Associative array with int as index:

```
  initial
  begin
    Ass_array = {10:2,5:7,1:8,3:9};

    $display(Ass_array);
    //first
    $display("searching for first element return value = %0d",Ass_array.first(Ass_array_index));
    $display("Ass_array_index of first = %d",Ass_array_index);
    //next
    $display("searching for next element return value = %0d",Ass_array.next(Ass_array_index));
    $display("Ass_array_index of next = %d",Ass_array_index);
    //last
    $display("searching for last element return value = %0d",Ass_array.last(Ass_array_index));
    $display("Ass_array_index of last = %d",Ass_array_index);
    //prev
    $display("searching for prev element return value = %0d",Ass_array.prev(Ass_array_index));
    $display("Ass_array_index of prev = %d",Ass_array_index);
  end
endmodule
```

Queue

- A queue is a variable-size, ordered collection of homogeneous elements.

Syntax : **<data_type>** **<name>[\$]** ;

int array[\$];

int array_1[0:\$];

Note: '0' representing the first and '\$' representing the last bound of Queue

- Sequential access to elements in the queue
- Insertion and removal at the beginning or the end of the queue.
- Can insert an element at any order in the queue using **insert()** method
- The maximum size of a queue can be limited by specifying its optional right bound (last index)
- Queue can be grows and shrinks automatically.

```
byte q1 [$];           // A queue of bytes
string names[$] = { "Bob" }; // A queue of strings with one element
integer q[$] = { 3, 2, 7 }; // An initialized queue of integers
bit q2[$:255];         // A queue whose maximum size is 256 bits
bit q2={ };            //empty queue
```

Queue Methods:

size():

size() method returns the number of items in the queue.

If the queue is empty, it returns 0.

syntax: **function int** size();

insert():

Inserts the given item at the specified index position.

syntax: **function void** insert (int index, queue_type item);

q.insert(i, e) is equivalent to
q = {q[0:i-1], e, q[i,\$]};

Example:

```
int q[$] = { 2, 4, 8 };  
initial begin  
$display(“size=%d”,q.size());  
end
```

o/p:
size = 3

Example: (ref above ex)

```
initial begin  
q.insert(2,10);  
$display(“%p”,q);  
end
```

o/p:
{2,4,10,8}

Contd..

delete()

deletes the element at the particular index

syntax: function void delete (int index);

q.delete (i) ;

q = {q[0:i-1], q[i+1,\$]};

Example:

```
int q[$] = { 2, 4, 8 };
```

```
    initial begin
```

```
        q.delete(1);
```

```
$display("queue %p",q);
```

```
end
```

o/p:

queue{2, 8}

Contd..

push_front();

Inserts the given element at the front of the queue.

syntax: function void

push_front(value);

q.push_front (value);

push_back();

Inserts the given element at the end of the queue.

syntax: function void push_back(value)

q.push_back (value);

Example:

initial begin

```
int q[$] = { 2, 4, 8 };
```

```
    q.push_front(1);
```

```
$display("push_front %p",q);
```

```
    q.push_back(10);
```

```
$display("push_back %p",q);
```

end

o/p:

```
push_front { 1,2,4,8}
```

```
push_back { 1,2,4,8,10}
```


Contd..

pop_back()

Removes and returns the last element of the queue.

syntax:

```
function queue_name pop_back();  
e = q.pop_back ( );
```

pop_front()

Removes and returns the first element of the queue.

syntax:

```
function queue_name pop_front();  
e = q.pop_front( );
```

Example:

initial begin

```
    int q[$] = { 7,9,2, 4, 8 };  
    q.pop_front();  
    $display("pop_front %p",q);  
    q.pop_back();  
    $display("pop_back %p",q);  
end
```

o/p:

```
pop_front {9,2,4,8}  
pop_back {2,4,8}
```

Queue with Locator Methods

- Can operate on any unpacked array, including queues, but their return type is a queue.
- It allow searching an array for elements (or their indexes) that satisfy a given expression.
- These can traverse the array in an unspecified order.

Syntax:

function array_type [\$] locator_method (array_type iterator = item); // same type as the array

Example with locator methods

```
int IA [*], qi [$];
```

```
IA[20]=7 ;      IA[40]=18 ;      IA[10]=27 ;
```

```
IA[60]=1 ;      IA[100]=18 ;      IA[300]=200 ;
```

```
qi =IA. find( x ) with ( x > 8 );
```

```
// Return a queue {27, 18,18 , 200}
```

```
qi =IA. find_index with ( item == 18 );
```

```
// Return a queue {40,100}
```

```
qi =IA. find_first with ( item >0 );
```

```
// Return a queue {27}
```

```
qi =IA. find_last( y ) with ( y <3 );
```

```
// Return a queue {1}
```

```
qi =IA. find_first_index with (item>15);
```

```
//Return a queue {10}
```

```
qi =IA. find_last_index(m) with (s>20);
```

```
//Return a queue {300}
```

```
qi =IA. min;
```

```
// Return a queue {1}
```

```
qi =IA. max;
```

```
// Return a queue {200}
```

```
qi =IA. unique;
```

```
// Return a queue {27,7,18,1,200}
```

Example with Ordering Methods

```
string s[] = { "hello", "sad", "world" };  
s.reverse;    // s becomes { "world", "sad", "hello" };  
logic [4:1] b = 4'bXZ01;  
b.reverse;    // b becomes 4'b10ZX  
Int q[$] { 4, 5, 3, 1 } ;  
q.rsrt;       // q become {5,4,3,1}  
int q[$] = { 4, 5, 3, 1 };  q.sort;           // q becomes { 1, 3, 4, 5 }
```

Example with Reduction Methods

```
byte b[$] = { 1, 2, 3, 4 };
```

```
int y;
```

```
y = b.sum ;
```

```
y = b.product ;
```

```
y = b.xor with ( item + 4 );
```

System Verilog (9.2) does not support any reduction methods other than sum.

```
// y becomes 10 => 1 + 2 + 3 + 4
```

```
// y becomes 24 => 1 * 2 * 3 * 4
```

```
// y becomes 12 => 5 ^ 6 ^ 7 ^ 8
```

```

module queues_array;
  //declaration
  bit  [31:0] queue_1[$];
  string queue_2[$];

  initial begin  //Queue Initialization:
    queue_1 = {0,1,2,3};
    queue_2 = {"Red","Blue","Green"};

    //Size-Method
    $display("----- Queue_1 size is %0d -----",queue_1.size());
    foreach(queue_1[i])
      $display("\tqueue_1[%0d] = %0d",i,queue_1[i]);
    $display("----- Queue_2 size is %0d -----",queue_2.size());
    foreach(queue_2[i])
      $display("\tqueue_2[%0d] = %0s",i,queue_2[i]);

    //Insert-Method
    queue_2.insert(1,"Orange");
    $display("----- Queue_2 size after inserting Orange is %0d -----",queue_2.size());
    foreach(queue_2[i])
      $display("\tqueue_2[%0d] = %0s",i,queue_2[i]);

    //Delete Method
    queue_2.delete(3);
    $display("----- Queue_2 size after Delete is %0d -----",queue_2.size());
    foreach(queue_2[i])
      $display("\tqueue_2[%0d] = %0s",i,queue_2[i]);
  end

endmodule

```

```

----- Queue_1 size is 4 -----
queue_1[0] = 0
queue_1[1] = 1
queue_1[2] = 2
queue_1[3] = 3
----- Queue_2 size is 3 -----
queue_2[0] = Red
queue_2[1] = Blue
queue_2[2] = Green
----- Queue_2 size after inserting Orange is
4 -----
queue_2[0] = Red
queue_2[1] = Orange
queue_2[2] = Blue
queue_2[3] = Green
----- Queue_2 size after Delete is 3 -----
queue_2[0] = Red
queue_2[1] = Orange
queue_2[2] = Blue

```

Comparison of Array

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 contiguous data**

Associate 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 analysing 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 contiguous data.**
- Memory usage is very good, as the size can be changed dynamically.

Queue

- No need of size information at compile time. It is automatically resized.
- **Performance to access elements is same as Static arrays.**
- Rich set of inbuilt methods for manipulating and analysing the content.
- Useful in self-checking modules , FIFO

KNOWING IS NOT ENOUGH;
WE MUST APPLY.

WILLING IS NOT ENOUGH;
WE MUST DO

Bruce lee

THANK YOU

Difference between Dynamic vs Associative vs queue

•Dynamic Array:

- We use dynamic array when we have no idea about the size of the array during compile time and we have to allocate its size for storage during run time.
- We basically use this array when we have to store a **contiguous** or **Sequential** collection of data.
- The array indexing should be always integer type.
- To allocate size of a dynamic array, we have to use `new[]` operator**

•Associative Array:

It is also allocated during run time.

- This is the array, where data stored in random fashion.
- It is used when we don't have to allocate contiguous collection of data, or data in a proper sequence or index.
- In associative array, **the index itself associates the data**. So it is called so.
- Indexing is not regular, can be accessed using indexing like integer or string type or any scalar.

•Queue:

Queue is a variable size, ordered collection of **Homogenous Data**.

- It is flexible, as it is variable in size and analogous to an **1-dimensional** Unpacked array that can shrink & grow automatically and can be of size zero.
- The main advantage of queue over dynamic array is that, we don't need **`new[]` operator** to allocate storage space for a queue.
- The other advantages of queue over dynamic array is that we can manipulate the queue using various queue methods like: **push, pop, delete, insert, size**.