1)

```
module dynamic_array();

  bit [7:0] dyn_a[];// ADD_CODE:Declare a dynamic array mem of 8 bits


initial begin
  dyn_a=new[4]; // ADD_CODE:Allocate the dynamic array for 4 locations
  $display ("Setting array size to 4");

  dyn_a={0,1,2,3};// ADD_CODE:Initialize the array with 0,1,2,3 values
  $display("Initialize the array with 0,1,2,3 values");
  foreach(dyn_a[i])
    begin
      $display("%d",dyn_a[i]);
    end

  dyn_a=new[8](dyn_a);  // ADD_CODE:Doubling the size of dynamic array, with old content still
valid
 //dyn_a.size;  // ADD_CODE:Display the current size of the dynamic array
  $display("size=%d",dyn_a.size);

  foreach(dyn_a[i])
    begin
      $display("dyn_a[%0d]=[%0d]",i,dyn_a[i]);
    end // ADD_CODE:Display the each value and the location of the dynamic array

 dyn_a.delete; // ADD_CODE:Delete all the elements in the dynamic array
 $display("size=%d",dyn_a.size);  // ADD_CODE:Display the current size of the dynamic array

   #1 $finish;
end

endmodule
```

Op

```
Setting array size to 4
Initialize the array with 0,1,2,3 values
0
1
2
3
size= 8
```

```
dyn_a[0]=[0]
dyn_a[1]=[1]
dyn_a[2]=[2]
dyn_a[3]=[3]
dyn_a[4]=[0]
dyn_a[5]=[0]
dyn_a[6]=[0]
dyn_a[7]=[0]
size= 0
```

2)

```
module associative_array ();

  int as_mem[int];
 // ADD_CODE:Declare an associative array as_mem of type int and index type int

int i;// ADD_CODE:Declare a local variable i of type int for manupilation


initial begin
  // ADD_CODE:Add element to the associative array as follows:
  as_mem[100] =101; // in the 100th location store value 101
  as_mem[1]=100; // in the first location store value 100
  as_mem[50]=99;// in the 50th location store value 99
  as_mem[256]= 77;// in the 256th location store value 77
  foreach(as_mem[i])
    begin
      $display("as_mem[%0d]=%0d",i,as_mem[i]);
    end
  // ADD_CODE:Display the size of the associative array
  $display("the size of the associative array",as_mem.size);
  // ADD_CODE:Check if index 2 exists
  as_mem.exists(2);
  $display(" checks if index 2 exists=%d", as_mem[2]);
  // ADD_CODE:Check if index 100 exists
  as_mem.exists(100);
  $display(" checks if index 100 exists=%d", as_mem[100]);
  // ADD_CODE: Display the value stored in first index
  as_mem.first(i);
  $display("the value stored in first index=%d",as_mem[1]);
```

```
  // ADD_CODE:Display the value stored in last index
  as_mem.last(i);
  $display("the value stored in last index=%d",as_mem[i]);
  // ADD_CODE:Delete the first index
  as_mem.delete(1);
  $display("Delete the first index=%d",as_mem[1]);
   foreach(as_mem[i])
    begin
      $display("as_mem[%0d]=%0d",i,as_mem[i]);
    end

  // ADD_CODE:Display the value stored in first index
  as_mem.first(i);
  $display("the value stored in first index=%d",as_mem[1]);
  #1 $finish;
end

Endmodule
```

Op

```
xcelium> run
as_mem[1]=100
as_mem[50]=99
as_mem[100]=101
as_mem[256]=77
the size of the associative array 4
checks if index 2 exists= 0
checks if index 100 exists= 101
the value stored in first index= 100
the value stored in last index= 77
Delete the first index= 0
as_mem[50]=99
as_mem[100]=101
as_mem[256]=77
the value stored in first index= 0
```

3)

```
program fork_join;
   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 );
 end
endprogram
```

Op

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

```
2 program fork_join_any;
 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 );
 end
```

```
  Endprogram
```

Op

```
[10] BEFORE fork  time =10
[15] time =15 # 5
[15] time = 15 Outside the main fork
```

3)

```
program fork_join_none;
  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 );
  end
endprogram
```

```
[10]  BEFORE fork  time = 10
[10]  time = 10 Outside the main fork .
```

```
module basic_data_types_simulation();
  // Declaring and initializing the variables
  bit [ 7:0] data = 8'b0101_01xz;
  logic [ 7:0] address = 8'b010z_01xz;
  integer write_addr = 32'b01x1_01xz_01xz_01xz;
  int read_addr = 32'b01x0_01xz_01xz_01xz;
```

```
  byte wr_enable = 8'b0101_01xz;
  reg rd_enable = 8'b0101_01xz;
initial
 begin

  // Displaying the values of the variables for different data types
  $display ("Showing outputs for different datatypes:\n");
  $display ("Value of bit data = %b\n", data);
  $display ("Value of logic address = %b\n", address);
  $display ("Value of integer write address = %b\n", write_addr);
  $display ("Value of int read address = %b\n", read_addr);
  $display ("Value of byte wr_enable = %b\n", wr_enable);
  $display ("value of reg rd_enable = %b\n", rd_enable);
  $display ("Output for bit + integer is = %b\n", data + address);
  $display ("Output for logic + int is = %b\n", write_addr + read_addr);

  // Re-assigning the variables for the different data types
  data = 10;
  address = 20;
  write_addr = 30;
  read_addr = 40;
  wr_enable = 16'habcx;
  rd_enable = 16'habcx;

  // Displaying the values of the variables for different data types after re-assigning
  $display ("After changing values, output for bit + logic is = %b\n", data + address);
  $display ("After changing values, output for integer + int is = %b\n", write_addr + read_addr);
  $display ("After changing values of byte wr_enable = %b\n", wr_enable);
  $display ("After changing values of reg rd_enable = %b\n", rd_enable);
 end
endmodule
Op
Showing outputs for different datatypes
Value of bit data =01010100
Value of logic address =010z01xz
Value of integer write address =00000000000000001x101xz01xz01xz
Value of int read address =00000000000000001000100010001000
Value of byte wr_enable =01010100
value of reg rd_enable=z
Output for bit + integer is = xxxxxxxx
Output for logic + int is=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 // Displaying the values of the variables for different data types after re-assigning
After changing values, output for bit + logic is =00011110
After changing values, output for integer + int is = 00000000000000000000000000100110
```

After changing values of byte wr_enable = 11000000
After changing values of reg rd_enable = x

```
module packed_and_unpacked();
// ADD_CODE:declare a packed array packed_array of 8 bits and initialize it to 8'hAA
  bit [7:0]packed_array ={8'hAA};
// ADD_CODE:declare an unpacked array unpacked_array of 8 bits and initialize it to
'{0,0,0,0,0,0,0,1}
  bit unpacked_array[8]='{0,0,0,0,0,0,0,1};

initial
 begin
  //ADD_CODE:display the 0th element of the packed array
   $display("the 0th element of the packed array=%d",packed_array[0]);

  //ADD_CODE:display the 0th element of the unpacked array
   $display("the 0th element of the packed array=%d",unpacked_array[0]);
  //ADD_CODE:display the whole packed array. Is it possible???
   $display("the whole packed array=%b",packed_array);
  //ADD_CODE:display the whole unpacked array. Is it possible???
   $display("the whole unpacked array=%p",unpacked_array);
  #1 $finish;
end

endmodule
```

op

```
the 0th element of the packed array=0
the 0th element of the packed array=0
the whole packed array=10101010
the whole unpacked array='{'h0, 'h0, 'h0, 'h0, 'h0, 'h0, 'h0, 'h1}
```

```
module queues();
// ADD_CODE:Declare a local variable i of type int for manupilation and initialize it to 1
  int i=1;
  int j;
```

```
// ADD_CODE:Declare a queue "b" of type int and initialize it to {3,4}
  int b[$]={3,4};

// ADD_CODE:Declare a queue "q" of type int and initialize it to {0,2,5}

  int q[$]={0,2,5};
initial
  begin
// ADD_CODE:Insert (1,j) into the queue q and display q using %p
    q.insert(1,"j");
    $display("size=%d",q.size);
    foreach(q[i])
      $display("q[%0p]=%p",i,q[i]);

// ADD_CODE:Insert (3,b[$]) into the queue q and display q using %p
    q.insert(3,b[$]);
    $display("size=%d",q.size);
    foreach(q[i])
      $display("q[%0p]=%p",i,q[i]);
// ADD_CODE:delete element (1) from the queue q and display q using %p
    q.delete(1);
     $display("size=%d",q.size);
    foreach(q[i])
      $display("q[%0p]=%p",i,q[i]);
// ADD_CODE:push front (6) into the queue q and display q using %p
    q.push_front (6);
     $display("size=%d",q.size);
    foreach(q[i])
      $display("q[%0p]=%p",i,q[i]);

// ADD_CODE:pop back from the queue q, store the value in j and display j
  j=  q.pop_back();
    $display("j=%d",j);

// ADD_CODE:push back (8) into the queue q and display q using %p
    q.push_back(8);
    foreach(q[i])
      $display("q[%0d]=%0p",i,q[i]);
// ADD_CODE:pop front from the queue q, store the value in j and display j
  j= q.pop_front();
    $display("j=%d",j);
end
```

endmodule

```
xcelium> run
size= 4
q[0]=0
q[1]=106
q[2]=2
q[3]=5
size= 5
q[0]=0
q[1]=106
q[2]=2
q[3]=4
q[4]=5
size= 4
q[0]=0
q[1]=2
q[2]=4
q[3]=5
size= 5
q[0]=6
q[1]=0
q[2]=2
q[3]=4
q[4]=5
j= 5
q[0]=6
q[1]=0
q[2]=2
q[3]=4
q[4]=8
j= 6
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

//Description: Concept of class inheritance and constructor with example
//*/**

```
//ADD_CODE: Write a class called base with property "value" of type int
  class base;
    int value;
//          explicitly override the class constructor - function new() in the class base
    function void display();
      //base b1=new();
//          and initialize the variable value to 3 inside the function new()
value=3;
     $display("value=%d",value);
endfunction
```

```
  endclass
//ADD_CODE: Write a class called ext which is extended from class base with property "x" of
type int
class ext extends base;
//          explicitly override the class constructor - function new() in the class ext

    int x=5;
     function void disp();
       $display("x=%d",x);
     endfunction
endclass
//          and initialize the variable "x" to 5 inside the function new()




program constructor1;
  initial
   begin
//ADD_CODE: Declare and create object for handle "e" of the class "ext"
     ext e=new();

//ADD_CODE: Display the variables "value" and "x" using the object "e"
     e.disp();
     e.display();
   end
endprogram
```

Op

```
xcelium> run
x= 5
value= 3
```

//Description: Concept of super in classes with example

```
//*************************************************************************/
//ADD_CODE: Write a class called parent with a task printf to display the message " THIS IS
PARENT CLASS "
class parent;
   task printf();
     $display("THIS IS PARENT CLASS ");
     endtask
```

```
      endclass
//ADD_CODE: Write a class called subclass which is extended from class parent.
      class subclass extends parent;
//         Write a task printf inside the class subclass and call the task printf of the class parent
using super
        task printf();
          super.printf();
        endtask
      endclass


program super1;

 initial
  begin
//ADD_CODE: Declare handle "s" for class subclass
  subclass s;


//ADD_CODE: Create object for handle "s"
    s=new();

//ADD_CODE: Call the task printf using object "s"
    s.printf();

  end
endprogram
```

Op

```
xcelium> run
THIS IS PARENT CLASS
```

//Description: Concept of polymorphism with example
//*/**

```
program polymorphism;
// ADD_CODE: Write a class called "Packet" with property "data" of 32 bits and
  class packet;
    bit [31:0] data;
//          function "send" with return type int and expecting argument "data"
//          of 32 bits. Inside the function "send" display "SENDING BASE PACKET"
```

```
   function int send(data);
     $display("SENDING BASE PACKET");
       return 0;

   endfunction
  endclass

// This is a Packet class, defining that there will
// be different types of packets to be sent

// ADD_CODE: Write a class called "Ethernet_packet" with property "ether_data" of 32 bits and
  class Ethernet_packet extends packet;
    bit [31:0]ether_data;
//          function "send" with return type int and expecting argument "ether_data"
//          of 32 bits. Inside the function "send" display "SENDING ETHERNET PACKET"
    function int send(ether_data);
      $display("SENDING ETHERNET PACKET") ;
       return 0;
    endfunction
  endclass

// ADD_CODE: Write a class called "Token_packet" with property "token_data" of 32 bits and
  class Token_packet extends packet;
    bit [31:0]token_data;
//          function "send" with return type int and expecting argument "token_data"
//          of 32 bits. Inside the function "send" display "SENDING ETHERNET PACKET"
function int send(token_data);
      $display("SENDING ETHERNET PACKET") ;
       return 0;
    endfunction
  endclass

//ADD_CODE: Declare an array of 10 handles for Packet class as pkts[10]
  packet pkts[10];

//ADD_CODE: Declare an handle for Ethernet_packet class as ep and
//          create object for it
  Ethernet_packet ep=new();

//ADD_CODE: Declare an handle for Token_packet class as tp and
//          create object for it
  Token_packet tp=new();

  bit [31:0]data;
```

```
initial
  begin
// ADD_CODE: Make the base class handles point to objects "ep" and "tp".


//          i.e. pkt[0]  points to ether packet and pkt[1] points to token packet

    pkts[0]=ep;
    pkts[1]=tp;
          // pkts[0].send(); is the same as calling ep.send(), but
          // the neat thing here is that a BFM only needs the
          // base class handle, and doesn't need to be modifed
          // if the functionality or data features change!!

// ADD_CODE: Call function "send" using handles pkts[0] and pkts[1],
//          Also pass the value for "data" in the function's argument list

    pkts[0].send(5);
    pkts[1].send(14);




end
endprogram
```

Op

```
xcelium> run
SENDING BASE PACKET
SENDING BASE PACKET
```

//Description: Concept of classes data type with example

```
//*************************************************************************/
//ADD_CODE: Declare a class called "simple" with properties i and j of int data type
class simple;
  int i;
  int j;
//          and write a task called printf to display the properties i and j of the class
  task printf();
```

```
   $display("i=%d,j=%d",i,j);
  endtask
endclass
program simple_class;
 initial
   begin
    //ADD_CODE: Declare two handles to the class simple as obj_1 and obj_2
    simple obj_1 , obj_2;
    //ADD_CODE: Create object for the handles obj_1 and obj_2
     obj_1=new();
      obj_2=new();
    //ADD_CODE: Access property i using obj_1 and initialize it to 2 and
    //        Access property i using obj_2 and initialize it to 4
    obj_1.i=2;
    obj_2.i=4;
    //ADD_CODE: Call the task printf using obj_1 and obj_2
     obj_1.printf();
     obj_2.printf();
   end
endprogram
```

Op

```
i= 2,j= 0
i= 4,j= 0
```

//Description: Concept of shallow copy with example

```
//*****************************************************************
*****/

program shallow_copy();

//ADD_CODE: Write a class "A" with property "j" of type int and
initialize it to 5
class A;
  int j=5;
endclass
//ADD_CODE: Write a class "B" with properties "i" of type int and
initialize it to 1
  class B;
    int i=1;
//          and declare handle "a" for class "A" and create object
for it
```

```
    A a=new();
  endclass

initial
  begin
//ADD_CODE: Declare a handle "b1" for class "B" and Create an object
for it
    B b1,b2;
//ADD_CODE: Declare a handle "b2" for class "B"

//ADD_CODE: Make a shallow copy of "b1" to "b2"
    b1=new();
    b2=new b1;

//ADD_CODE: Display "b1.i, b2.i, b1.a.j, b2.a.j"
    $display("b1.i=%d, b2.i=%d, b1.a.j=%d, b2.a.j=%d",b1.i, b2.i,
b1.a.j, b2.a.j);
//ADD_CODE: Now change the value of "i" in "b2" to 10
  b2.i=10;

//ADD_CODE: Display "b1.i, b2.i, b1.a.j, b2.a.j"
   $display("b1.i=%d, b2.i=%d, b1.a.j=%d, b2.a.j=%d",b1.i, b2.i,
b1.a.j, b2.a.j);

//ADD_CODE: Now change the value of "j" in "b2.a" to 50
 b2.a.j=50;
$display("b1.i=%d, b2.i=%d, b1.a.j=%d, b2.a.j=%d",b1.i, b2.i, b1.a.j,
b2.a.j);

//ADD_CODE: Display "b1.i, b2.i, b1.a.j, b2.a.j"


  end

endprogram


Op
b1.i= 1, b2.i= 1, b1.a.j= 5, b2.a.j= 5
b1.i= 1, b2.i= 10, b1.a.j= 5, b2.a.j= 5
b1.i= 1, b2.i= 10, b1.a.j= 50, b2.a.j= 50
```

At what time clock is equal to 1 for the code below

```
Initial
Begin
Clk =0;
#5
Fork
#5 a = 0;
#10 b = 0;
Join
Clk= 1;
end
```

ans:AT time=15, clk=1.

```
module queues();
// ADD_CODE:Declare a local variable i of type int for
manupilation and initialize it to 1
int i=1;
  int j;
// ADD_CODE:Declare a queue "b" of type int and
initialize it to {3,4}
  int b[$]={3,4};
// ADD_CODE:Declare a queue "q" of type int and
initialize it to {0,2,5}

  int q[$]={0,2,5};
```

```
initial
  begin
// ADD_CODE:Insert (1,j) into the queue q and display
q using %p
    q.insert(1,"j");
    $display("q=%p",q);

// ADD_CODE:Insert (3,b[$]) into the queue q and
display q using %p
    q.insert(3,b[$]);
    $display("q=%p",q);
// ADD_CODE:delete element (1) from the queue q and
display q using %p
    q.delete(1);
    $display("q=%p",q);
// ADD_CODE:push front (6) into the queue q and
display q using %p
    q.push_front(6);
    $display("q=%p",q);
// ADD_CODE:pop back from the queue q, store the value
in j and display j
   j= q.pop_back;
    $display("j=%p",j);
// ADD_CODE:push back (8) into the queue q and display
q using %p
    q.push_back(8);
    $display("q=%p",q);
// ADD_CODE:pop front from the queue q, store the
value in j and display j
    j= q.pop_front;
    $display("j=%p",j);
end

endmodule
```

```
// Code your testbench here
// or browse Examples
module associative_array ();

// ADD_CODE:Declare an associative array as_mem of
type int and index type int
  int as_mem[int];
// ADD_CODE:Declare a local variable i of type int for
manupilation
int i;

initial begin
  // ADD_CODE:Add element to the associative array as
follows:
  as_mem[100]=101; // in the 100th location store
value 101
  as_mem[0]=100;  // in the first location store value
100
  as_mem[50]=99;// in the 50th location store value 99
  as_mem[256] =77;// in the 256th location store value
77
  $display("as_mem=%p",as_mem);
  // ADD_CODE:Display the size of the associative
array

  $display("the size of the associative array",
as_mem.size);
  // ADD_CODE:Check if index 2 exists
  $display("Check if index 2
exists",as_mem.exists(2));
  // ADD_CODE:Check if index 100 exists
  $display("Check if index 100
exists",as_mem.exists(100));
  // ADD_CODE: Display the value stored in first index
```

```
  as_mem.first(i);
  $display("the value stored in first
index",as_mem[i]);
  // ADD_CODE:Display the value stored in last index
  as_mem.last(i);
  $display("the value stored in last
index",as_mem[i]);
  // ADD_CODE:Delete the first index

  // ADD_CODE:Display the value stored in first index
  as_mem.first(i);
  as_mem.delete(i);
  $display("value stored in first index",as_mem[i]);

end
endmodule
```

op

```
as_mem='{1:100, 50:99, 100:101, 256:77 }
# the size of the associative array 4
# Check if index 2 exists 0
# Check if index 100 exists 1
# the value stored in first index 100
# the value stored in last index 77
# ** Warning: (vsim-3829) Non-existent associative array entry.
Returning default value.
# Time: 0 ns Iteration: 0 Instance: /associative_array File:
testbench.sv Line: 34
# value stored in first index 0
```