LAB-III

Objective: - The objectives of this lab are: -

- Randomization
- Inter-process communication
- System Verilog assertions

Experiment I: - (Randomization): System Verilog support randomization of variables, class properties. SV provides randomize () method and rand keyword to support randomization.
Random Variables: - The variables declared with "rand" keyword are random variables.

```
Ex: - rand bit [3:0] data;
```

Another keyword system Verilog support for randomization is "randc". "randc" is a random-cyclic. The variables declared using "randc" won't get the repeat values until every possible value has been assigned. To randomize the object variables, we have to call randomize() method.
Ex: - object.randomize ();

Example:

```
class packet;
rand bit [3:0] data1;
randc bit [3:0] data2;
endclass

module randomization;
initial begin
packet packet1;
packet1 = new();
repeat (32) begin
packet1.randomize();
$display("\tdata1 = %0d \t data2 = %0d",packet1.data1,packet1.data2);
end
end
endmodule
```

Constraint Randomization: - Random variables can be constrained and variables can get specific value. The constraints to random variable are written in separate constraint blocks. Constraint blocks are class members like tasks, functions and variables. The constraint blocks defined outside the class like extern methods are known as extern constraint blocks.

```
Ex: - constraint data_range { data > 5;} //the variable data will get a
value greater than 5.
```

```
class packet;
rand bit [3:0] data1;
randc bit [3:0] data2;
constraint data_range { data1 > 5;}
endclass
module constraints;
initial begin
packet packet1;
packet1 = new ();
repeat (16) begin
packet1.randomize();
$display("\tdata1 = %0d",packet1.data1);
end
end
endmodule
```

Random variables can be constrained and variables can get specific value. The constraints will get inherited from parent class to child class. The rules of class inheritance are applicable to random variables in classes.

```
Example: - (correct this program)
class packet;
rand bit [3:0] data1;
randc bit [3:0] data2;
constraint data_range { data1 > 16;}
endclass

class packet1 extends packet;
constraint data_range { data1 < 16; }
endclass

module random_inherit;
initial begin
packet pack1;
packet1 pack2;

pack1 = new ();
pack2 = new ();
    repeat(16) begin
      pack1.randomize();
      $display("\tpkt1:: data1 = %0d",pack1.data1);
    end


    repeat(16) begin
      pack2.randomize();
      $display("\tpkt2:: data1 = %0d",pack1.data1);
    end
   end
endmodule
```

Keyword “inside” in the randomization constraint: -

```
Example: -
constraint data_range { data_range inside {1,3,4,5,7};}
constraint data_range { data_range inside {1,3,[5:10],18, [21:25]};}


class packet;
rand bit [3:0] data1;
randc bit [3:0] data2;
randc bit [3:0] data3;
constraint data1_range {data1 inside {1,3,5,7,9,11,13,15};}
endclass

module constraints;
initial begin
packet packet1;
packet1 = new ();
repeat (5) begin
packet1.randomize();
$display("\tdata1 = %0d",packet1.data1);
end
end
endmodule
```

Weighted Distribution constraint: The random variables can be assigned with some values repeatedly or verification engineer can control the values of randomized variables by applying weighted distribution. The "dist" operator takes a list of values and weights and values assigned to random variables can be tightly controlled.

```
Ex: - value := weight

data dist { 2 := 5, [10:12] := 8 };

for data == 2 , weight 5
    data == 10, weight 8
    data == 11, weight 8
    data == 12, weight 8

Value :/ weight/n

addr dist { 2 :/ 5, [10:12] :/ 8 };

for data == 2 , weight 5
    data == 10, weight 8/3
    data == 11, weight 8/3
    data == 12, weight 8/3
```

In the above syntax: -
Value: - is a desired value to a random variable.
Weight: - defines the probability of value getting assigned to random variable.
Default weight of unspecified value is: =1 and sum of weights need not be 100. The weighted distribution is specified to the values inside the constraint block. The probability of value with more weight getting assigned to random variable is high.

```
Example: -
class packet;
  rand bit [3:0] data_1;
  rand bit [3:0] data_2;

  constraint data_1_range {   data_1 dist { 2 := 5, [10:12] := 8 }; }
  constraint data_2_range {   data_2 dist { 2 :/ 5, [10:12] :/ 8 }; }
endclass

module constr_dist;
  initial begin
    packet packet1;
    packet1 = new();


    repeat(32) begin
      packet1.randomize();
      $display("\tdata_1 = %0d",packet1.data_1);
    end

    repeat(32) begin
      packet1.randomize();
      $display("\tdata_2 = %0d",packet1.data_2);
    end

  end
endmodule
```

Functions in random constraints:

```
class packet;
  rand bit [7:0] start_addr;
  rand bit [7:0] end_addr;

  constraint end_addr_c { end_addr == e_addr(start_addr); }

  function bit [7:0] e_addr(bit [7:0] s_addr);
    if(s_addr<0)
      e_addr = 0;
    else
        e_addr = s_addr * 3;
  endfunction
endclass

module func_constr;
  initial begin
    packet packet1;
    packet1 = new();

    repeat(8) begin
      packet1.randomize();
      $display("\tstart_addr = %0d
end_addr=",packet1.start_addr,packet1.end_addr);
    end
  end
endmodule
```

Unique constraints in randomization: SV support “unique” keyword in randomization and variables defined with this keyword will get unique values during randomization. This is applicable to variables and array elements. In the below example, the three different variables can be obtained by using unique keyword.

Example: -

```
class unique_elements;
  rand bit [3:0] a,b,c;
  rand bit [7:0] array[6];
  constraint varis_c {unique {a,b,c};}
  constraint array_c {unique {array};}

  function void display();
    $display("a = %p",a);
    $display("b = %p",b);
    $display("c = %p",c);
    $display("array = %p",array);
  endfunction
endclass

program unique_elements_randomization;
  unique_elements packet1;

  initial begin
    packet1 = new();
    packet1.randomize();
    packet1.display();
  end
endprogram
```

Solve Before: - Solve before is used in the constraint block to specify the order of constraint solving. This property is helpful when variables are dependent.

```
Example: -

class packet;
  rand bit       x;
  rand bit [3:0] y;

  constraint sxy { solve x before y;}
  constraint x_y { (x == 1) -> y == 0;}
endclass

module inline_constr;
  initial begin
    packet packet;
    packet = new();
    repeat(10) begin
      packet.randomize();
      $display("\tValue of x = %0d, y = %0d",packet.x,packet.y);
    end
  end
endmodule
```

Bi-directional Constraints: In this feature of SV, all random variables are solved parallel. The variable ‘a’ is a sum of ‘b’ and ‘c’. The constraint solver will solve the value of ‘a’ to satisfy the constraints on ‘b’ and ‘c’.

```
class packet;
  rand bit [3:0] a;
  rand bit [3:0] b;
  rand bit [3:0] c;

  constraint a_value { a == b + c; }
  constraint b_value { b > 6; }
  constraint c_value { c < 8; }
endclass

module bidirec_constr_random;
  initial begin
    packet packet1;
    packet1 = new();
    repeat(5) begin
      packet1.randomize();
      $display("Value of a = %0d \tb = %0d \tc
=%0d",packet1.a,packet1.b,packet1.c);
    end
  end
endmodule
```

Experiment 2 (Inter-process Communication)

This experiment comprises of semaphores and mailbox.

Semaphore: - SV support semaphores for access control to shared resources and synchronization.

Semaphore methods: - SV support built-in classes for semaphores: -

1. new (): create semaphores with a specified number of keys.

```
Example: semaphore_name = new (no. of keys);
```

2. get (): get a key or keys to semaphore.

```
Ex: -semaphore_name.put () or semaphore_name.put(no. of keys);
```

3. put(): return the semaphore key or keys

```
Ex: - semaphore_name.get () or semaphore_name.get (no. of keys);
```

4. try_get (): Try to obtain the key or keys without blocking.

```
Example: -
module semaphore1;
  semaphore sem;
  initial begin
    sem=new(1);
    fork
      display();
      display();
    join
  end

  task automatic display();
    sem.get();
    $display($time,"\tCurrent Simulation Time");
    #30;
    sem.put();
  endtask
endmodule

Example: -
module semaphore2;
  semaphore sem;

  initial begin
    sem=new(4);
    fork
      display(2);
      display(3);
      display(2);
      display(1);
    join
  end

  task automatic display(int key);
    sem.get(key);
    $display($time,"\tCurrent Simulation Time, Got %0d keys",key);
    #30;
    sem.put(key);
  endtask
endmodule
```

Mailbox: - Mailbox is a communication structure used to exchange the message between the processes. Mailbox in SV can be categorized as: - bounded mailbox and unbounded mailbox. In bounded mailbox is with size defined and size of unbounded mailbox is unlimited. There are two types of mailboxes: generic mailbox and parameterized mailbox.
Example: -

```
mailbox  mailbox_name;
```

Mailbox methods: SV supports following mailbox methods:-
new (); //create a mailbox
put (); // place a message in a mailbox
try_put (); //Try to place a message in mailbox without blocking
get (); or peek (); // retrieve a message from a mailbox
num (); //returns the number of messages in mailbox.
try_get (); or try_peek (); // try to pull a message from mailbox without blocking.

Example: - Mailbox is used to communicate between generator and driver. Generator class will generate the packet and put into mailbox and driver class access the generated packet from the mailbox.

```
class packet;
  rand bit [7:0] address;
  rand bit [7:0] data;

  function void post_randomize();
    $display("Packet::Packet Generated");
    $display("Packet::address=%0d,Data=%0d",address,data);
  endfunction
endclass

class generator;
  packet packet1;
  mailbox m_box;

  function new(mailbox m_box);
    this.m_box = m_box;
  endfunction
  task run;
    repeat(2) begin
      packet1 = new();
      packet1.randomize();
      m_box.put(packet1);
      $display("Generator::Packet Put into Mailbox");
      #5;
    end
  endtask
endclass

class driver;
  packet packet1;
  mailbox m_box;


  function new(mailbox m_box);
    this.m_box = m_box;
  endfunction
```

```
  task run;
    repeat(2) begin
      m_box.get(packet1);
      $display("Driver::Packet Recived");

$display("Driver::address=%0d,Data=%0d\n",packet1.address,packet1.data);
    end
  endtask
endclass

module mailbox_ex;
  generator gen;
  driver    dri;
  mailbox m_box;
  initial begin
    m_box = new();
    gen = new(m_box);
    dri = new(m_box);

    fork
      gen.run();
      dri.run();
    join

  end
endmodule
```

Events: - are useful for synchronization of processes. Event operation has got two parts. First part trigger the event and second part wait for event to be triggered. Events are triggered using the operator -> and ->>. The wait for an event can be triggered using @ operator or wait ().

```
module events;
  event ev_1;
  initial begin
    fork

      begin
        #40;
        $display($time,"\tTriggering The Event");
        ->ev_1;
      end

      begin
        $display($time,"\tWaiting for the Event to trigger");
        #60;
        @(ev_1.triggered);
        $display($time,"\tEvent triggered");
      end
    join
  end
  initial begin
    #100;
    $display($time,"\tEnding the Simulation");
    $finish;
  end
endmodule
```

Experiment 3 (Assertions): Assertions are used to verify the behavior of a design. Assertions are useful in checking the sequence of events and conditions. Broadly, assertions can be categorized as: - immediate assertions and concurrent assertions.
Immediate assertion: - Immediate assertions verify the conditions at the current simulation time.

```
Ex: - label: assert (expression) action_block;

Ex: - always@(posedge clk) assert (a && b);

module asertion1;
  bit clk,x,y;
  always #5 clk = ~clk;

  initial begin
    x=1;
    y=1;
    #15 y=0;
    #10 y=1;
        x=0;
    #20 x=1;
    #10;
    $finish;
  end

  always @(posedge clk) assert (x && y);

endmodule
```

Concurrent Assertions: - Concurrent assertions verify the sequence of events spread over multiple clock cycles. Concurrent assertions are evaluated at the occurrence of every clock tick. Concurrent assertion can be placed in procedural block or in a module or in an interface.

```
Ex: con_assert: assert property (@(posedge clk) not (a @@ b));
```

Concurrent assertion will have: - Boolean, sequences, property and property directive layer.

```
Example: -

module concur_asse(
  input wire clk,req1,reset,
  output reg grant);

// Sequence Layer

sequence req1_grant_seq;
  (~req1 & grant) ##1 (~req1 & ~grant);
endsequence

// Property Layer

property req1_grant_prop;
  @ (posedge clk)
    disable iff (reset)
      req1 |-> req1_grant_seq;
endproperty

// Assertion Directive Layer

req1_grant_assert : assert property (req1_grant_prop)
                 else
```

```
                $display("@%0dns Assertion Fail", $time);

// Sample DUT RTL

always @ (posedge clk)
  grant <= req1;

endmodule

//   Testbench Code

module concur_asse_tb();

reg clk = 0;
reg reset, req1 = 0;
wire grant;

always #4 clk ++;

initial begin
  reset <= 1;
  #30 reset <= 0;
    #100 @ (posedge clk) req1  <= 1;
  @ (posedge clk) req1 <= 0;

  #100 @ (posedge clk) req1  <= 1;
  repeat (5) @ (posedge clk);
  req1 <= 0;
  #10 $finish;
end

concurrent_assertion dut (clk,req1,reset,grant);

endmodule
```

Building SV Assertions: - SVA checkers can be developed as shown below: -
Step 1: Create Boolean expression: Functionality is described in this section and it could be a simple Boolean equation. Boolean layer is lowest in the layer and Boolean checking is either true or false. If the variables in the equation contain x or Z, the it results in false.
Step 2: Create Sequences: In this layer, we use Boolean layer to construct the valid sequence of events. SVA provide a keyword “sequence” to define these events. Sequences can be declared in: - module, interface, program, clocking block.
Step 3: Create Property: The numbers of sequences are combined to create the complex sequences. Such sequences are called property. The SVA support a keyword “property” to support this feature.
Step 4: Assert Property: SVA support a keyword “assert” to check the property. The property is verified during simulation.

The sequences play a major role in developing accurate assertions. The list of operators used in sequences is: -
Delay (##): - ex: req ##1 gnt; // gnt happens after the one clock tick after req.
Repetition [*]: - specifies the number of repetitions. `Ex: - req ##1 gnt [ * 2]`

Example: -

```
module repetition_ass();

logic clk = 0;
always #1 clk ++;

logic request,busy,grant;

// Sequence Layer

sequence boring_way_seq;
  request ##1 busy ##1 busy ##1 grant;
endsequence

sequence cool_way_seq;
  request ##1 busy [*2] ##1 grant;
endsequence

sequence range_seq;
  request ##1 busy [*1:5] ##1 grant;
endsequence


// Property Specification Layer

property boring_way_prop;
  @ (posedge clk)
      request |-> boring_way_seq;
endproperty

property cool_way_prop;
  @ (posedge clk)
      request |-> cool_way_seq;
endproperty

// Assertion Directive Layer

boring_way_assert : assert property (boring_way_prop);
cool_way_assert   : assert property (cool_way_prop);


// Generate input vectors

initial begin
  request <= 0; busy <= 0;grant <= 0;
  @ (posedge clk);
  request <= 1;
  @ (posedge clk);
  busy <= 1;
  request  <= 0;
  repeat(2) @ (posedge clk);
  busy <= 0;
  grant <= 1;
  @ (posedge clk);
  grant <= 0;
  request <= 0; busy <= 0;grant <= 0;
  @ (posedge clk);
  request <= 1;
  @ (posedge clk);
  busy <= 1;
  request  <= 0;
```

```
  repeat(3) @ (posedge clk);
  busy <= 0;
  grant <= 1;
  @ (posedge clk);
  grant <= 0;
  #50 $finish;
end

endmodule
```

SVA methods: SVA supports following methods: -
$rose: - returns true, when LSB gets logic 1.
$fell: - returns true, when LSB gets logic 0.
$stable: - returns true, when the value of a related variable not changed since last clock tick to current clock tick.
$past: - returns number of clock ticks (n).

Example (SVA methods): -

```
module system_assertion();

logic clk = 0;
always #4 clk ++;

logic request,grant;

// Property Specification Layer

property system_prop;
  @ (posedge clk)
      ($rose(request) && $past(!request,1)) |=>
         ($rose(grant) && $past(!grant,1));
endproperty

// Assertion  Layer

system_assert : assert property (system_prop);

initial begin
  request <= 0;grant <= 0;
  repeat(10) @ (posedge clk);
  request <= 1;
  @( posedge clk);
  grant <= 1;
  request <= 0;
  @( posedge clk);
  // Make the assertion fail now
  request <= 0;grant <= 0;
  repeat(10) @ (posedge clk);
  request <= 1;
  @( posedge clk);
  request <= 0;
  grant <= 0;

  #30 $finish;
end

endmodule
```

Different Types of Assertion Clocking:

Clock is mandatory for concurrent assertions and it clocking for assertions can be achieved in different ways as described below: -

- A sequence with a clock instance
- A property with a clock instance
- Clock from procedural clock
- Clocking block

```
Example: -

module clock_resolve_assertion();
logic clk = 0;
logic request,grant;

// Clock inside a sequence

sequence request_grant_seq;
  @ (posedge clk)
      request ##1 grant;
endsequence

// Clock inside a property

property request_grant_prop;
  @ (posedge clk)
      request |=> grant;
endproperty

// Clock infered from a always block

always @ (posedge clk)
begin
   grant <= request;

   request_grant_assert : assert property (request  |=> grant);
end

// Default clocking

default clocking aclk @ (posedge clk);
  input request;
  input grant;
endclocking

property request_grant_default_prop;
   request |-> ##1 grant;
endproperty

// clocking clocking

clocking requestclk @ (posedge clk);
  input request;
  input grant;
endclocking

property request_grant_clocking_prop;
   requestclk.request |-> ##1 requestclk.grant;
endproperty
```

```
a1  : assert property (request_grant_prop);
a2  : assert property (request_grant_default_prop);
a3  : assert property (request_grant_clocking_prop);


always #4 clk ++;

initial begin
  $monitor("request %b grant %b",request,grant);
  request <= 0; grant <= 0;

  ##1 request  <= 1;
  ##20;
  request <= 0;
  #10 $finish;
end

endmodule
```

Implication Property: is similar to if_else. Implication checks for the preceding sequence to occur to check behavior. Left hand side is known as antecedent and right hand side is called as consequent. If antecedent succeeds, then the consequent is evaluated.

There are two types of implication operator: -
|-> (Overlapped implication): - In overlapped implication, if antecedent succeeds, then consequent expression is evaluated in the same clock.
|=> (Non-overlapped implication): - The consequent expression is evaluated in the next clock cycle, if antecedent succeeds.

```
Example:

module implicat_ass();

logic clk = 0;
always #1 clk ++;

logic request,busy,grant;

// Sequence Layer

sequence implication_seq;
  request ##1 (busy [->3]) ##1 grant;
endsequence


// Property Specification Layer

property overlap_prop;
  @ (posedge clk)
      request |-> implication_seq;
endproperty

property nonoverlap_prop;
  @ (posedge clk)
      request |=> implication_seq;
endproperty

// Assertion Directive Layer
```

```
overlap_assert    : assert property (overlap_prop);
nonoverlap_assert : assert property (nonoverlap_prop);


// Generate input vectors

initial begin

  gen_seq(3,0);
  repeat (20) @ (posedge clk);

  gen_seq(3,1);

  #45 $finish;
end

task  gen_seq (int busy_delay,int grant_delay);
  request <= 0; busy <= 0;grant <= 0;
  @ (posedge clk);
  request <= 1;
  @ (posedge clk);
  request  <= 0;
  repeat (busy_delay) begin
   @ (posedge clk);
    busy <= 1;
   @ (posedge clk);
    busy <= 0;
  end
  repeat (grant_delay) @ (posedge clk);
  grant <= 1;
  @ (posedge clk);
  grant <= 0;
endtask

endmodule
```

Experiment 4 (Functional Coverage): Functional Coverage is a metric to measure how much of the design specification has been exercised by testbench. There are two types of functional coverage: - Data oriented coverage and Control oriented coverage.
Data oriented coverage: - Check the data values occurred on I/O and variables sufficiently. This type of coverage can be achieved by writing coverage groups, coverage points and cross coverage.
Control oriented coverage: - This type of coverage can be achieved by checking the sequence of behavior has occurred. This coverage can be achieved by writing assertions.

Example of Function Coverage: -

```
module simple_coverage();

logic [7:0]  address;
logic [7:0]  data;
logic        par;
logic        rw;
logic        en;

// Coverage Group

covergroup memory @ (posedge en);
  addressess : coverpoint address {
    bins low    = {0,50};
    bins med    = {51,150};
    bins high   = {151,255};
  }
  parity : coverpoint  par {
    bins even  = {0};
    bins odd   = {1};
  }
  read_write : coverpoint rw {
    bins  read  = {0};
    bins  write = {1};
  }
endgroup

// Instance of covergroup memory

memory mem = new();

// Task to drive values

task drive (input [7:0] a, input [7:0] d, input r);
  #5 en <= 1;
  address  <= a;
  rw    <= r;
  data  <= d;
  par   <= ^d;
  $display ("@%2tns address :%d data %x, rw %x, parity %x",
     $time,a,d,r, ^d);
  #5    en <= 0;
  rw    <= 0;
  data  <= 0;
  par   <= 0;
  address  <= 0;
  rw    <= 0;
endtask

// Testvector generation

initial begin
  en = 0;
  repeat (10) begin
    drive ($random,$random,$random);
  end
  #10 $finish;
end

endmodule
```

References: -

1. www.verificationguide.com
2. www.testbench.in
3. Synopsys VCS user guide version 2016
4. Cadence NC-Verilog User manual