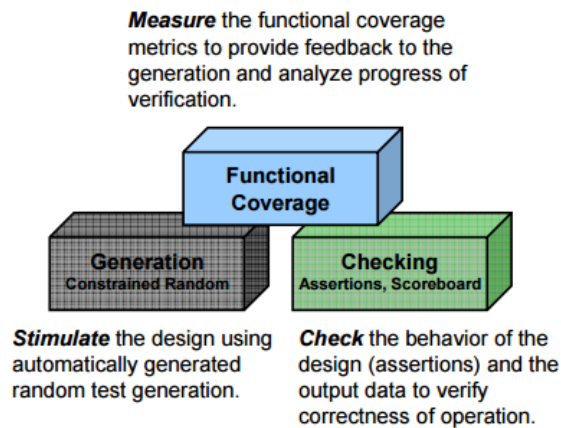


SYSTEM VERILOG

(For Verification)

Chapter 1: Verification Overview

The System Verilog verification methodology relies on three building blocks:



Randomization: Automatic random generation of stimulus. Range and probability of values can be controlled via constraints: **Conditional, Distributed and Weighted**. System Verilog supports randomization of simple variables or complex object-oriented data ☐ Also additional constructs for randomization of events and sequence of data.

Assertions (SVA): Used to specify and verify behavior of a design ☐ Can be added anywhere in a design hierarchy ☐ Ignored by synthesis tools ☐ Checked during simulation ☐ Can also be used to provide functional coverage ☐ Can also be used as an input to formal verification tools.

Functional coverage: Coverage allows the user to tell how well a design has been tested. System Verilog has two types of functional coverage:

◆ **Data-oriented coverage**

- ☐ Coverage groups, coverage points and cross products
- ☐ Checks combinations of data values have occurred

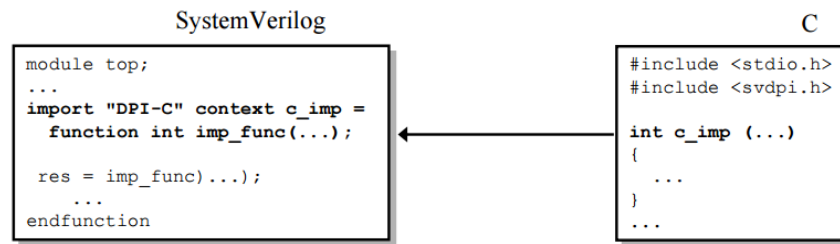
```
// SV coverage group:
covergroup cg1 @(posedge clk);
  Addr: coverpoint addr
  { bins low  = { [0:'h0F], 19 };
    bins mid[] = { 16, 17, 18 };
    bins high = { ['h14:'hFF] }; }
  AddrXvalid : cross Addr, valid;
endgroup
cg1 cover_inst = new;
```

◆ **Sequence-oriented coverage**

- ☐ Using SystemVerilog Assertions (SVA)
- ☐ Checks whether sequences of behaviors have occurred

```
// SVA functional coverage
property full2empty;
  @(posedge clk)
    fifo_full ##[1:$] fifo_empty;
endproperty
C1 : cover property (full2empty);
```

Direct Programming interface: Is a new lightweight C-programming interface. Maps between System Verilog subprograms and C routines. Without the overhead associated with the PLI: System Verilog can call subprograms implemented in C and C can call routines implemented in System Verilog



Chapter 2: Simple verification features

Topics

- 1) Strings
- 2) Immediate assertions
- 3) Fork-join enhancements
- 4) Design features useful for verification

- 1) System Verilog provides a string type, which is essentially a dynamic array of byte elements with some methods defined to manipulate it as a character string. You can still do the operations you would expect to do on a vector reg that happens to hold the ASCII representation of a character string. Just remember that indexing a string is selecting a byte instead of a single bit. Verilog provides escaped character sequences for including newline (\n), tab (\t), backslash (\\) and quote (\") characters, and any octal representation of a character, within a string literal. To these, System Verilog adds the form feed (\f), vertical tab (\v) and bell characters (\a), and any hexadecimal representation of a character (\x0A).

```

string message = "test ";

if (pass)
    message = {message, "passed"};
else
    message = {message, "failed"};
$display("%s", message);
    
```

```

string repstr;

repstr = {2{"go "}};
    
```

String operators:

Operator	Usage	Function	Description
==	s1 == s1	Equality	Returns 1 if strings are equal
!=	s1 != s2	Inequality	Logical negation of ==
< <= > =>	s1 < s2 s1 <= s2 s1 > s2 s1 >= s2	Comparison	Return 1 if condition is true.
{Str,Str}	s = {s1,s2,s3}	Concatenation	Returns concatenated string
{N{Str}}	s = {5{s1}}	Replication	Returns string replicated N times
[]	b = s1[index]	Indexing	Returns the byte at index

String methods:

Method	Description	Syntax & Usage
len()	Returns length of the string (length excludes any terminating character)	function int len() str.len();
putc()	Replaces i th character of the string with the integral value c	task putc(int i, byte c) str.putc(i, c);
getc()	Returns ASCII code of the i th character in the string	function byte getc(int i) x = str.getc(i);
compare()	Compares strings with regard to lexical ordering	function int compare(string s) str.compare(s); // compares str to s
icompare()	Compares strings with regards to lexical ordering and is sensitive to character case	function int icompare(string s) str.icompare(s); // compares str to s
substr()	Returns new string that is a substring formed by characters in position i through j of str.	function string substr(int i, int j) str.substr(i,j);

Method	Description	Syntax & Usage
toupper() tolower()	Returns the upper/lower case of the specified string. String is unchanged.	function string toupper () x = str.toupper();
atobin() atooct() atoi() atohex()	Returns integer value corresponding to ASCII binary / octal / decimal / hexadecimal representation. Scans all leading digits and underscore (_) characters and stops when any other characters are encountered.	function int atobin() str = "21334"; int i = str.atoi(); // i = 21334;
atoreal()	Returns real value corresponding to ASCII decimal representation of a number	function real atoreal() str = "3.1416"; real r = str.atoreal(); // r = 3.1416;
bintoa() octtoa() itoa() hextoa()	Stores the ASCII representation of a binary / octal / decimal / hexadecimal number into a string	task hextoa (int i) i = 16'h5356; str.hextoa(i); // str = "5356";
realtoa()	Stores the ASCII representation of a real number into the string	task realtoa(real r) r = 3.1416; str.realtoa(r); // str = "3.1416";

- 2) Procedural statement, similar to an if statement. It may be labeled, the access to the label is via %m formatter. When executed evaluate a Boolean expression and success only if evaluates to 1. It reports error message by default, you can change this behavior.

```
always @(negedge clock)
  A1: assert !(wr_en && rd_en);
```

Action block: You can provide an “action” block to execute upon success, failure or both. The action block may override the default reporting behavior.

```
always @(negedge clock)
  rw_chk: assert !(wr_en && rd_en)
    $display ("%m: success");
  else
  begin
    $display("read/write fail");
    err_count++;
    -> rw_err_event;
  end
```

Severity levels: \$info, \$warning, \$error (default), \$fatal (terminates simulation). Severity level is reported. You can append additional information, using syntax identical to that for \$display.

Immediate assertions: provide only a non-temporal Boolean check Extra code required – as if we had used if: Timing controls and Loops. (the ones used before).

Comparison between assertions types:

An immediate assertion is an instantaneous boolean check

- ◆ Single-cycle

```
always @(posedge CLK iff CE)
begin : CHECK
  repeat (16)
  begin
    assert ( CE && SCLK )
    else // short pulse
    ...
    @(posedge CLK);
    assert ( CE && !SCLK )
    else // short pulse
    ...
    @(posedge CLK);
  end
  assert ( !CE && !SCLK )
  else // long pulse
    $error("long CE pulse");
end : CHECK
```

SystemVerilog also has *concurrent* assertions

- ◆ Can span multiple cycles

```
sequence SCYC;
  (CE && SCLK) ##1 (CE && !SCLK);
endsequence

SPI1 : assert property ( @(posedge CLK)
!CE ##1 CE |-> SCYC[*16] ##1 !CE );
```

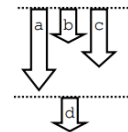
3)

Verilog-2001 `fork...join` completes when all spawned blocks complete

- ◆ This blocks further execution until the `fork...join` completes

Verilog fork-join

```
fork
  a;b;c;
join
  d;
```

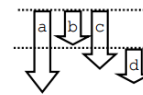


SystemVerilog adds two join variants to control when `fork...join` completes

- ◆ `join_any` completes the fork as soon as *any* of the blocks complete
 - ❑ Other blocks left running
- ◆ `join_none` completes the fork *immediately*
 - ❑ All blocks left running

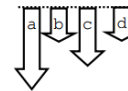
SystemVerilog

```
fork
  a;b;c;
join_any
  d;
```



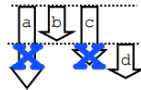
SystemVerilog

```
fork
  a;b;c;
join_none
  d;
```



Fork sizable and wait fork statements:

```
fork
  a;b;c;
join_any
  disable fork;
  d;
```

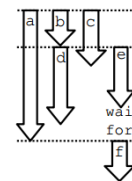


```
fork
  a;b;c;
join_any

fork
  d;
join_none

e;

wait fork;
f;
```



Test example:

The test bench loops through the array, applying stimulus, waiting one clock cycle, and checking the response. It checks the response by using an immediate assertion, which upon failure, displays the message associated with that stimulus and response pair.

```

module LUTtest;
  parameter size_of_array = 20;
  import mytypes::*;
  logic ip1, ip2, ip3; // DUT inputs
  logic op1, op2, op3, op4, op5, op6; // DUT outputs
  ... // instantiate DUT
  stimresp test_data[(size_of_array-1):0];
  wire [5:0] resp; // expected results
  assign resp = {op1, op2, op3, op4, op5, op6};

  initial //      Stimulus Response Message
    test_data = { {3'b000, 6'b100000, "init error"},
                  {3'b001, 6'b000010, "t1 failed"},
                  ... };

  initial begin
    @(negedge clk);
    for (int i = (size_of_array-1); i >= 0; i--) begin
      {ip1, ip2, ip3} <= test_data[i].stim;
      @(negedge clk);
      assert (resp == test_data[i].resp)
        else $error(test_data[i].mess);
    end
  end
  ...

```

```

package mytypes;
  typedef struct {
    bit[2:0] stim;
    bit[5:0] resp;
    string mess;
  } stimresp;
endpackage

```

Unions: You use a union to store different data types at the same memory location. The union size is the size of its largest field.

```

union {
  logic [31:0] udata; // unsigned data
  integer sdata;      // signed data
} inbus, outbus;      // 32-bit variables

inbus.udata = 32'h00f3;
outbus.sdata = -44;

```

```

typedef union {
  int i;
  shortreal f;
} num; // named union type

num n1, n2; // instances of union type

n1.f = 0.0; // set n1 in floating point format
n2.i = -44; // set n2 in integer format

```

The first example declares an array of a named structure type that contains a field that is a union. The other structure field is a bit that indicates whether the structure union field is currently being used as an int type or as a float type. System Verilog provides the tagged union qualifier to serve exactly that purpose. A tagged union stores both its current value and its current tag. Subsequent expressions accessing the tagged value must be of the current tag type. This type checking is generally done during run time.

```

typedef struct {
  union {
    int i;
    shortreal f;
  } n; // anonymous union type
  bit isfloat;
} tagged_s; // named struct type

tagged_s a[9:0]; //array of structure containing union

```

```

typedef union tagged {
  int i;
  shortreal f;
} tagged_u;

tagged_u value; int j; shortreal g;
value = tagged f 3.14156;
g = value.f; // legal
j = value.i; // illegal

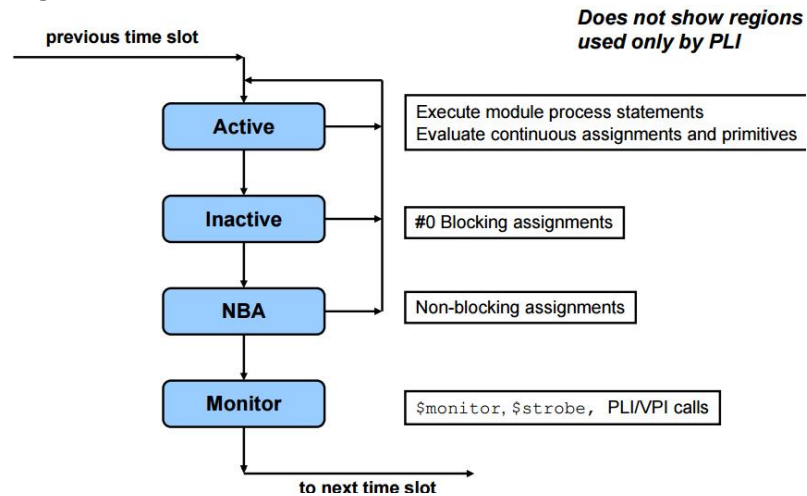
```

Chapter 3: Verification Blocks

Topics

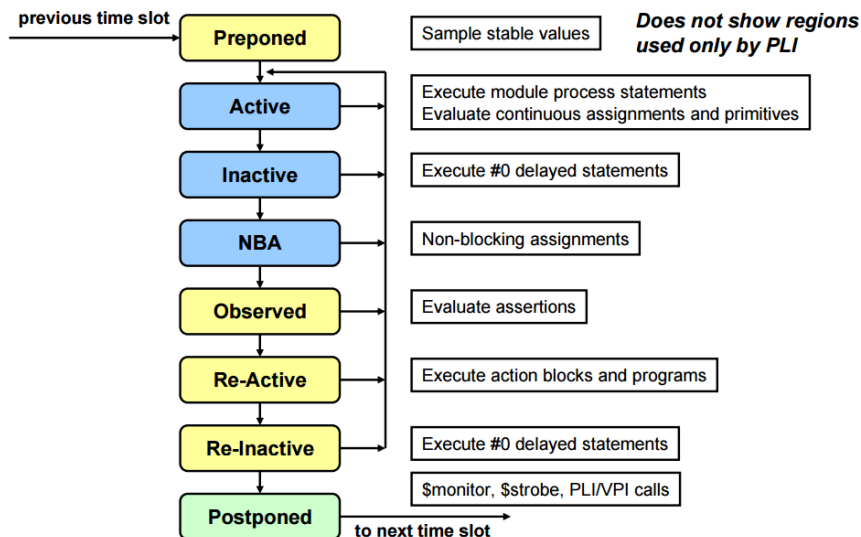
- 1) Event scheduler
- 2) Final blocks
- 3) Programs
- 4) locking blocks

1) Simplified Verilog event scheduler:



The first three of these regions are iterative – they can schedule events that require return to the Active region. When no more events exist for the current simulation time, the simulator executes Monitor statements and then advances simulation time to the next time for which events are scheduled. The simulation terminates when no such future events exist.

System Verilog event scheduler:



System Verilog adds regions to provide a predictable interaction between assertions, design code and test bench code. It adds the Observed region in which to evaluate assertions, and the Re-Active and Re-Inactive regions in which to execute assertion action blocks and test bench programs.

- 2) A final procedural block executes at the end of simulation. Executes after explicit or implicit call to \$finish. Execute exactly once. Cannot consume simulation time (no blocking delays). You can use to calculate and display simulation statistics. System Verilog executes the final blocks upon encountering \$finish or upon running out of events to process. Any final block that itself calls \$finish or calls a user-defined system task or function that terminates the simulation aborts

execution of the final blocks. After System Verilog completes or terminates execution of the final blocks, it calls any PLI routines you have scheduled for execution at the end of the simulation.

```
final begin
  if (timeout_error)
    $display ("ERROR: %0t: Test Timed Out", $time);
  else
    $display ("INFO: %0t: Test Complete", $time);
    $display("Error Count: %d", error_count);
    $display("Fifo Overflow Count: %d", fifo_overflow);
end
```

- 3) Programs are somewhat similar to modules, but are intended for the test-bench: To clearly separate design and test, cannot have an always block. cannot instantiate hierarchy. Execute in the reactive region of the time slot. Can use \$exit system task.

```
program memtest (
  output wire [7:0] data;
  output bit [4:0] addr;
  output bit read, write);
  ...
  initial begin
    ...
  end
endprogram : memtest
```

It is intended to remove test-bench stimulus from the module construct and instead execute those statements in the REACTIVE region separately from the execution and update of design code. To further reduce the potential for user-induced clock/data races, the program construct requires the user to make only blocking assignments to program variables and only non-blocking assignments to module and interface variables.

Illegal constructs in programs: always blocks, anything associated with hierarchy, parameters overrides (defparam), specify blocks, specparams declarations.

Use blocking assignment (=) to update program variables (Any declared local to a program). Use non-blocking assignment (<=) to update non-program variables.

```
module design (logic dA);
  logic dB;
  int dC;
endmodule : design
```

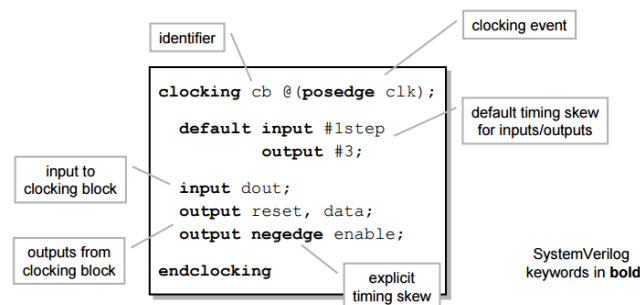
```
module top;
  logic top_A;
  design d1 (top_A);
  test t1 (top_A);
endmodule: top
```

```
program test (logic tA);
  logic temp;
  int number;
  initial begin
    tA  <= 0; // ERROR
    temp <= 1; // ERROR
    number = 15; // okay
    top.d1.dB = 0; // ERROR
    top.d1.dC <= 5; // okay
  end
endprogram : test
```

Only programs can reference program variables. Nothing outside a program can reference program variables. On the other hand, Program can reference outside variables. Can call \$exit only from a program. Terminates all processes in the current program: initial blocks and their sub-processes, not continuous assignments and not tasks called from other programs. \$finish called when all program blocks have exited (Either through \$exit or normally).

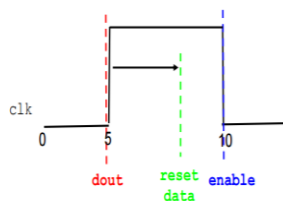
You can define anonymous programs in packages and in compilation-unit scopes. Anonymous programs can declare a restricted set of program items without declaring a new scope. You do not need to instantiate nested programs with no ports or top-level programs. If you do not explicitly instantiate these, they are implicitly instantiated once with an instance name that is the same as their definition name.

- 4) A clocking block defines a set of timing, relative to a specified clock, for a set of signals. Any number of signals may appear in any number of clocking blocks. You declare a clocking block as an interface, module or program item. You can declare one clocking block in any scope to be the default clocking block for that scope. You apply the timing by referencing the signal hierarchically through the clocking block name.



A clocking block is both a declaration and instance of that declaration. You do not instantiate a clocking block. Input skew designates sample time for signal before the clocking event: Defaults to #1step, Output skew designates driving time for signal after the clocking event, Defaults to #0. You can specify skew: As a default for inputs, as a default for outputs, explicitly with the signal identifier, overriding the default. as an edge, number or time literal o Number uses current timescale

Clocking Skew Example



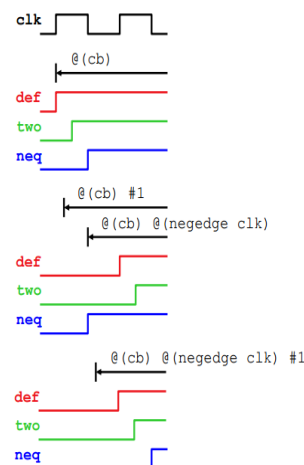
```

clocking cb @(posedge clk);
default input #1step
    output #3;
input dout;
output reset, data;
output negedge enable;
endclocking
  
```

Assume clock period of 10ns, timescale of 1ns and timeprecision of 10fs

`dout` sampled #1step before (posedge clk) = 10fs before 5ns
`reset, data` driven #3 after (posedge clk) = 8ns
`enable` driven on (negedge clk) = 10ns

More Output Skews



```

var bit def=0,two=0,neg=0;

clocking cb @(posedge clk);
output def;
output #2 two;
output negedge neg;
endclocking

initial
begin
    ... // timing control
    cb.def<=1;
    cb.two<=1;
    cb.neg<=1;
end
  
```

You can define multiple clocking blocks in a scope. For multiple clocks, different signals, or simply different timing. Only one block in a scope can be the default clocking block. Add default

to the beginning of the clocking block declaration. Or use a default statement separate from the declaration.

```
clocking cb1 @(posedge clk1);
default input #2 output #3;
input cdout;
output reset, data;
endclocking

clocking si1 @(posedge clk2);
input #2 ip1;
output #2 op1;
endclocking

default clocking si2 @(posedge clk2);
input #2 ip2;
output #5 op2;
endclocking

clocking si2 @(posedge clk2);
input #2 ip2;
output #5 op2;
endclocking

default clocking si2;
```

To apply the timing of a clocking block, you hierarchically reference the signal through the clocking block name. To synchronize a process to a clocking block, you can either: Directly use the clocking block name in a sensitivity list. This triggers the sensitive process on the clocking block clock event. Use clocking block signal inputs, or slices of them, in a sensitivity list. This triggers the sensitive process on the clocking block signal event.

To apply timing, access the signal through the clocking block

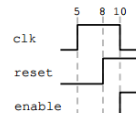
- ◆ cb.enable <= 1;
 - Must use only non-blocking assignments to clocking block signals
- ◆ dreg <= cb.dout;
- ◆ @(cb) // @(posedge clk)
- ◆ @(cb.dout) // sampled values

```
bit reset, enable;
clocking cb @(posedge clk);
default input #1step output #3;
input dout;
output reset, data;
output negedge enable;
endclocking

initial begin
  @(cb);
  cb.reset <= 1'b1;
  cb.enable <= 1'b1;
  @(cb.dout);
```



Timing applies only when signal accessed through clocking block



The semantics of the cycle delay (##N) operator differ depending upon how you use it: **When used as a procedural or intra-assignment delay** in an assignment to a clocking block signal, it refers to the clock event of the specified clocking block. These assignments must use the non-blocking operator. **When not used as a procedural or intra-assignment delay** in an assignment to a clocking block signal, it refers to the default clocking block. For this situation, the compiler shall issue an error if no default clocking block has been declared.

```
default clocking def @(negedge clk);
endclocking
clocking cb @(posedge clk);
input #1step dout;
output #3 reset, data;
endclocking

// Wait 2 def cycles
@(def);
@(def);
// Wait 2 default clocking cycles
##2;

// Drive data after 1 cb cycle
##1 cb.data <= 2'b01;

// drive data after 1 def cycle
##1; cb.data <= 2'b10;

// Drive with current tmpdat value
// after 3 cb cycles
cb.data <= ##3 tmpdat;
```



A clocking block signal cannot be a hierarchical expression You can associate a clocking block signal with a hierarchical expression.

```
clocking cb @(posedge clk);
  default input #1step
    output #3;
  input dout;
  //data associated with hierarchical expression
  output reset, data = top.dut.data; // Valid
  // output top.dut.data;           // Invalid
  output negedge enable;
endclocking
```

Chapter 4: Transaction Level Modeling (TLM)

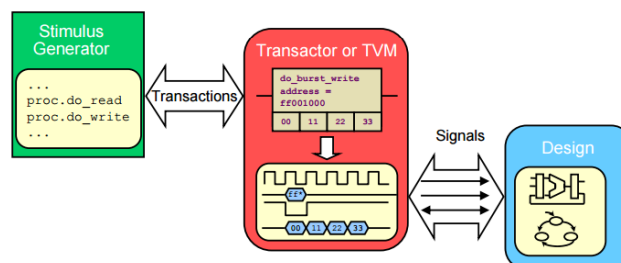
Topics

- 1) What is a Transaction?
- 2) Transaction Based Verification (TBV)
- 3) Transactors
- 4) Transaction structure
- 5) Interface review
- 6) Interface transactors

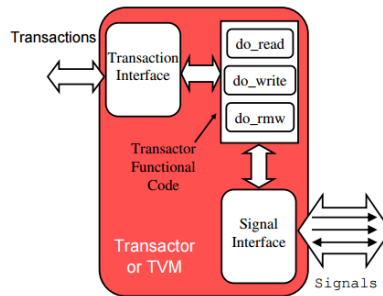
- 1) A transaction is a transfer of data between design blocks. A transfer may be composed of many signal transitions over several cycles. Transfers are easier to create, debug, maintain and manage. Transactions may have parameters (address, data, etc.). Transactions are nothing new ☐ Used in script-driven test-benches.

Transactions are most useful when you have well-defined interfaces for your design and test environment. Interface methods implement the transactions.

- 2) A Transaction-Based Verification (TBV) environment contains two layers: ☐ The transaction producer generates a stream of data and operations to be performed on the data. ☐ The Transaction Verification Model (TVM), also called a transactor, converts between the test environment transactions and the low-level signal transitions at the Design Under Verification (DUV). This conversion can go both ways, in which case the test environment also consumes transactions.



- 3) A transactor has three parts: Transaction interface (Stimulus generator -transaction producer- and Response checker -transaction consumer-), Signal interface (DUV ports), and Transactor functionality (Create data sequences to drive signal interface and Checks signal interface sequences for correct operation).



- 4) **Simple transactor using interfaces:** An interface can implement most of the transactor functionality. Here is a simple interface: It has a clock port, it declares design signals, Interface tasks (methods) control the design signals. Hence it is “transactor”.

```
interface spi_if (input clk);

    logic sdi, sdo;
    logic ce;

    task init ();
    begin
        ce = 0;
        sdi = 0;
        @(posedge clk);
    end
    endtask

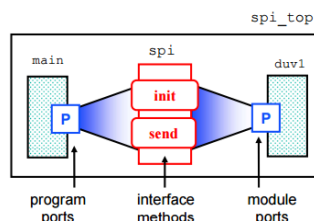
    task send(input [7:0] waddr,
              input [7:0] wdata );
    begin
        sdi = 0;
        ce = 1;
        @(posedge clk);
        for (int i = 15; i>0; i--)
            ... // write serializer
        end
    endtask
endinterface
```

Driving Transaction for program block: For more complex interfaces and transactions, you may find it easier to verify the design with direct access to the design ports rather than relying solely upon interface methods. This permits more control over the design stimulus, for example, you can craft fork...join blocks “on-the-fly” that probably do not exist in the regular reusable interface methods. Most System Verilog test-benches utilize this flexible approach, using a combination of interface methods and program functionality.

```
interface spi_if(input clk);
    logic sdi, sdo;
    logic ce;
    task init ();
    ...
    task send (...);
    ...
endinterface
```

```
program spi_test (spi_if spi);
...
    initial begin
        spi.init;
        spi.send(add, data);
        spi.ce <= 0;
        ...
    end
endprogram
```

```
module spi_top;
    logic clock;
    spi_if spi(clk); // interface
    spi_test main(spi); // program
    duv duv1 (spi); // module
endmodule
```



- 5) You can define a modport to prevent the DUV from accessing the transaction interface:

```
interface spi_if(input clk);

    logic sdi, sdo;
    logic ce;

    // interface modports
    modport tb ( output sdi, ce, clk,
                 input sdo,
                 import init, send );

    modport duv ( input sdi, ce, clk,
                 output sdo );

    task init ();

program spi_test (spi_if.tb spi);
...
    initial begin
        spi.init;
        spi.send(add, data);
        ...
    end

module duv (spi_if.duv main);
...
    always @ ( posedge main.clk)
    ...
endmodule
```

You can place a clocking block in an interface. Interface methods can drive signals via the clocking block. The clocking block can specify the modport signal directions. Directions are from perspective of program utilizing the modport. You would not typically use a clocking block for a module that represents design behavior.

```
interface spi_if(input clk);

    logic sdi, sdo;
    logic ce;

    default clocking cb1 @(posedge clk);
    default input #1 output #3;
    input sdo;
    output ce, sdi;
    endclocking

    // interface modports
    modport tb (clocking cb1,
                 import init, send);

    modport duv (input sdi, ce, clk,
                 output sdo);

    task init ();
    begin
        cb1.ce <= 0;
        cb1.sdi <= 0;
        #1;
    end
    endtask
```

- 6) A virtual interface is an interface variable that at various times throughout the test can represent different interface instances. By iterating through an array of interface variables and passing each element in turn to a test task, your one test-bench task can test multiple instances of similar design blocks, each through its own interface instance. You can declare virtual interfaces in modules, programs, classes and packages. You can pass virtual interfaces by value, but not by reference, to tasks and functions.

```
// Interface
interface inv_if;
    var logic in_var, out_var;
endinterface
```

```
// Inverter
module inv(interface intf);
    assign intf.out_var = !intf.in_var;
endmodule
```

```
module top;
    inv_if intf1(), intf2();
    inv inv1(.intfc(intf1));
    inv inv2(.intfc(intf2));
    virtual inv_if intf;
    task check (exp);
        ...
    endtask
    task do_test;
        intf.in_var = 0;
        #1 check(1);
        intf.in_var = 1;
        #1 check(0);
    endtask
    initial begin
        intf = intf1; do_test;
        intf = intf2; do_test;
        $display("TEST DONE");
        $finish(0);
    end
endmodule
```

Benefits of TBV: You can verify designs at transaction level instead of signal level. Allowing you to think and debug at a higher level of abstraction, thus more quickly develop and analyze the test environment. Rather than wading through a “sea of waveforms” at signal level. You can partition test development into **stimulus** and **transactors**. This isolates the test development from the design development. Stimulus changes do not necessarily require design changes. Design changes do not necessarily require stimulus changes. This methodology is most effective for designs with well-defined interfaces

Chapter 5: Classes

Topics

- 1) Class overview
- 2) Properties and instances
- 3) Constructors and methods
- 4) Static properties and methods
- 5) Aggregates and Inheritance
- 6) Information hiding
- 7) Virtual classes
- 8) Polymorphism

- 1) A class is a user-defined type, that declares data variables, and methods to manipulate the data, and that you can dynamically create and delete during the simulation. You use classes for such things as transactions, where during simulation, an undetermined number may come into being and then disappear.

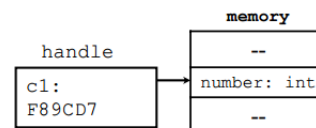
```
class myClass;
  int number;
endclass
```

An object is an instance of a class. Declare it as a variable (handle). “Points” to the memory representing the instance (initial value null). Create it using function. Object persists until either: Handle assigned null or handle not accessible.

```
class myClass;
  int number;
endclass

myClass c1; // variable = null
c1 = new; // object created

myClass c2 = new; // option
...
c1 = null; // delete object
```



A class is a user data type. To use a class, you declare a variable of the class type. The variable is a handle that “points” to a location in memory that holds the class properties. An uninitialized handle has a null value. You create a class instance by calling a special class method called “new”. The “new” method is a function that allocates an area of memory for the class properties and places a pointer to that area in the class object handle variable.

System Verilog restricts what you can do with object handles. Note that System Verilog performs automatic memory management!

Operation	C++ Pointer	SystemVerilog Object Handle
Arithmetic operations, e.g. incrementing	allowed	not allowed
For arbitrary data types	allowed	not allowed
Assignment to an address of a data type	allowed	not allowed
Dereferencing	allowed	not allowed
Casting	unlimited	limited
Default value	0	null
Memory management	manual	automatic

- 2) The class type declaration also declares class members. The members can be variables, called “properties”, and they can also be tasks or functions, called “methods”, to manipulate the class properties. You reference the members of a class object by using the “dot” (.) hierarchy separator.

<pre>class myClass; int number; task set (input int i); number = i; endtask function int get; return number; endfunction endclass</pre>	<div>direct access</div> <pre>myClass c1 = new; initial begin c1.number = 4; \$display("c1: %d", c1.number); end</pre>	<div>method access</div> <pre>myClass c2 = new; initial begin c2.set(3); \$display("c2: %d", c2.get()); end</pre>
---	--	---

Classes offer much more than structs, so for test-benches classes are preferred.

External method declaration:

<pre>class myClass; int number; task set (input int i); number = i; endtask extern function int get; endclass function int myClass::get; return number; endfunction</pre>	<pre>myClass c2 = new; initial begin c2.set(3); \$display("c2: %d", c2.get()); end</pre>
---	--

3) Class construct:

default constructor	explicit constructor	explicit constructor with parameter
<pre>class defcon; int number; endclass defcon c1 = new; // c1.number = 0</pre>	<pre>class expcon; int number; function new(); number = 5; endfunction endclass expcon c2 = new; // c2.number = 5</pre>	<pre>class parcon; int number; function new(input int init); number = init; endfunction endclass parcon c3 = new(5); // c3.number = 5</pre>

Properties of 2-state types initially take on the 0 value and 4-state types take on the unknown value.

- 4) Each class instance has its own unique copy of a “normal” property. Only one copy of a static property exists, and all class instances access that same copy, in fact, you can access that copy through the class name without having any instances of the class. <class_name::static_member>

Class Example

```
class frame;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity = 0;

  function new (input int add, dat);
    addr = add;
    payload = dat;
    parity = par({addr, payload});
  endfunction

  function bit par (input logic[12:0] ip);
    return (^ip);
  endfunction

  function logic [13:0] getframe();
    return({addr, payload, parity});
  endfunction
endclass
```

Note:

- ◆ Initial property value
- ◆ Constructor parameter values applied to properties
- ◆ Constructor calls method to get updated parity value
- ◆ Application repeatedly calls method to serialize frame

```
frame one = new(addr1, data1);
frame two = new(3, 4);

for (int i=0; i<14; i++) begin
  @(negedge clk);
  serial = one.getframe()[i];
end
```

Static Properties

```
class frame;
  static int frmcount=0;
  int tag;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity;

  function new(input int add, dat);
    addr = add;
    payload = dat;
    parity = par({addr, payload});
    tag = ++frmcount;
  endfunction
...
endclass
```

Exactly one instance of a **static** property always exists, not associated with any objects

This example uses a static property to provide a unique identifier for each new frame

- ◆ Initially 0
- ◆ Incremented by constructor

frame one = new(1, 0);

frame two = new(3, 2);

one	
tag	1
addr	1
payload	0
parity	1

frmcount: 1

two	
tag	2
addr	3
payload	2
parity	1

frmcount: 2

Only static methods can manipulate static properties, and they cannot manipulate anything other than static properties. You can access static methods through any class instance, or through the class name. <class_name::static_member>

An example, based on the previous description:

```
static function int getcount();
  return (frmcount);
endfunction
```

```
frame one = new(addr1, data1);
frame two = new(3, 4);
one = null;
two = null;
no_frms = frame::getcount(); // 2
```

The **this** keyword is a handle to the current class object. You can use it to reference class identifiers re-declared within a local scope. Meaningful only for non-static members This example uses the fully qualified class property “this.addr” in a function that re-declares the “addr” identifier.

```
class frame;
  static int frmcount;
  int tag;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity;

  function new(input int addr, dat);
    this.addr = addr;
    payload = dat;
    parity = par({addr, payload});
    tag = ++frmcount;
  endfunction
endclass
```

Your class type declaration can itself declare handle variables of other class types. This is known as an aggregate class type. In this example, the tagframe class type declaration declares a handle variable of the frame class type. The tagframe constructor calls the frame constructor to initialize the frame object within the tagframe object. You can say that the tagframe object “has” a frame object, much as a truck “has” an engine.

```

class frame;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity;

  function new(input int add, dat);
    addr = add;
    payload = dat;
    parity = par({addr, payload});
  endfunction
  ...
endclass

class tagframe;
  frame f1;
  static int frmcount;
  int tag;

  function new(input int add, dat);
    f1 = new(add, dat);
    tag = ++frmcount;
  endfunction
  ...
endclass

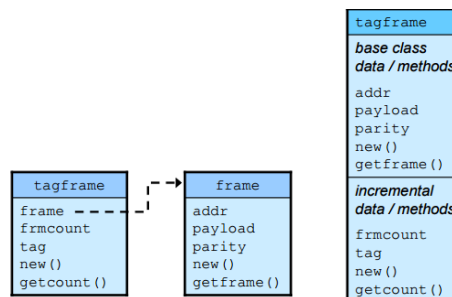
tagframe one = new(add1, dat1);
tagframe two = new(3, 4);

one.f1.addr = 0;

```

5) **Aggregation:** A class type instantiates the handles of other class types – relationship “has a”.
Containing class constructor initializes handles

Inheritance: A class type inherits the members of another class type – relationship “is a”. The parent class type declaration is known as the superclass or base class, and the extended child type declaration is known as the subclass or derived class. System Verilog supports Inheritance of only a single parent class type.



Simple inheritance: Subclass inherits superclass members: Can add more members, can re-declare superclass methods. A subclass type is the superclass type – syntax to access members is identical. Subclass constructor automatically calls superclass constructor. For this example, entry into tagframe.new() immediately implicitly calls frame.new().

```

class frame;
  logic [4:0] addr;
  logic [7:0] payload;
  bit parity;

  function new(input int add, dat);
    addr = add;
    payload = dat;
    parity = par({addr, payload});
  endfunction
  ...
endclass

class tagframe extends frame;
  static int frmcount;
  int tag;

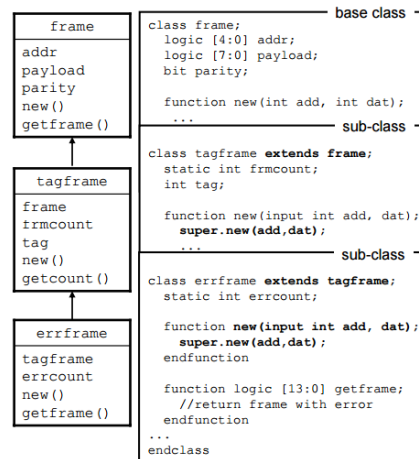
  function new(input int add, dat);
    super.new(add, dat);
    tag = ++frmcount;
  endfunction
  ...
endclass

tagframe one = new; //???
one.addr = 0;

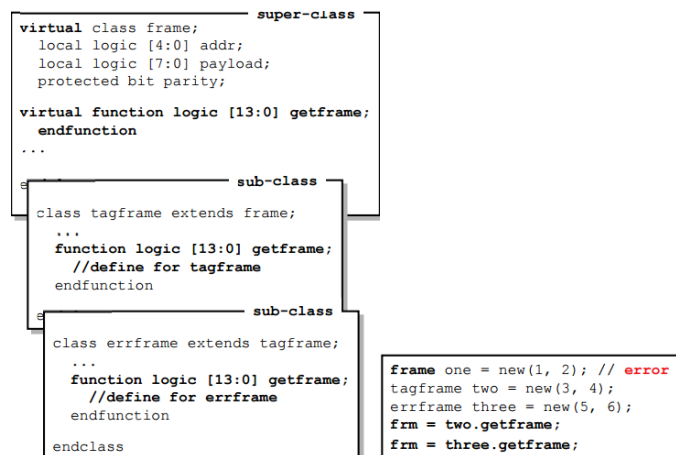
```

The superclass (frame) constructor takes parameters. The subclass (tagframe) constructor must on the first line pass these parameters to its super class.

Multiple layer inheritance: You can layer inheritance to multiple generations. A tagframe object is a frame object that also has a tag and a frame count. A errframe object is a tagframe object that has an error count and provides a corrupt frame.

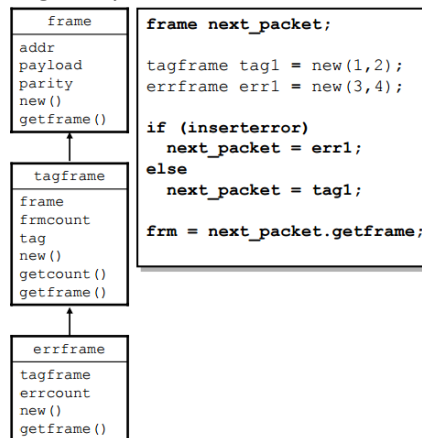


- 6) **Hiding information:** when we refer to a member, it could possible mean a property or a method. All members are by default “public” – accessible externally. Use the **local** keyword to make a member visible only to class methods. Use the **protected** keyword to make a member visible only to class methods and methods of subclasses. Finally, you can use the **const** keyword, which defined the member as read-only, it could a global const (the default value is specified in the declaration) or an instance const (the value is defined inside the constructor).
- 7) A virtual class exists only as a superclass – you cannot instantiate it. Only a virtual class may have virtual methods. Prototype only – no implementation, subclass provides implementation.



- 8) You can use a handle of a superclass type to “point” to objects of its subclasses. Remember that any of the subclass type declarations can redeclare members of its superclass, thus creating a hierarchy of member method implementations. For a “normal” method, the compiler selects which version to use based upon the handle type, so a handle of the superclass type, even if it “points” to a subclass object, accesses the methods declared in the superclass type. For a virtual method, the compiler creates a look-up table, so that during simulation, the method implementation is selected based upon the subclass type of the object that the superclass handle happens to “point” to during the method call, so a handle of the superclass type, if it “points” to

a subclass object, accesses the methods declared in that subclass type. This mechanism is called “polymorphism”, literally, “having many forms”.



Parameterized classes: You can parameterize a class declaration just like you parameterize an interface, module, or program. Parameters can be values or types – hence “template class”. You can override parameter values and types on a per-instance basis. Methods using type parameters must work for all expected type overrides. Each combination of parameter types has its own set of any static variables.

```

class stack #(parameter type st = int);
    local st data[100];
    static int depth;
  
```

Chapter 6: Random stimulus

Topics

- 1) Random and directed stimulus
- 2) Pseudo-random number generators and seeding
- 3) Unsigned random numbers
- 4) Scope randomization
- 5) Constraints
- 6) Randcase
- 7) Randsequence

- 1) Structural metrics, consisting of cover-points and cover crosses in a cover-group, looks good when every possible value to each variable has been applied. Functional metrics, consisting of cover statements with design properties, do not look good if we do that. We would miss most of the transitions between values. What we really want to do is to generate random sequences of operations, registers and data, and constrain those random combinations to meaningful combinations.

With randomization, you can with minimal testbench code produce large amounts of stimulus that thoroughly and effectively verifies the design. With constraints, you can restrict the generated values and value transitions to those that are meaningful. SystemVerilog offers two forms of randomization:

- Randomization of local variables
- Randomization of class properties

2) Pure random number sequences are useless for verification, sequence must be repeatable, since it allows debug-fix-debug cycle. Verification uses pseudo-random sequences, controlled by seed(s): Same seed always gives the same sequence of numbers and different seed gives a different sequence.

3) Unsigned random numbers:

- \$urandom(seed): generates a 32-bit random unsigned integer, seed is optional.
- \$urandom_range(maxval,minval): generates a 32-bit random unsigned integer within a range, the order of the values specified in the range can be altered.

4) You can randomize singular variables of any integral type: Pass variables to function std::randomize(). Returns 1 on success (variables can randomize and constraints can be met). Otherwise returns 0.

```
//type declarations as before
opcode_t  opcode;
regs_t    regs;
logic[7:0] data;
int ok;

for (int i = 0; i<7168; i++)
begin
    ok = randomize(opcode, regs, data);
    @(posedge clk);
end...
```

To maintain random stability, add new threads and class objects and randomization calls after previous threads and class objects and randomization calls, or explicitly initialize the RNGs. You can guarantee random stability by using the srandom() method to manually seed the Random Number Generator (RNG) of the process.



Tip

self() is a static function of the built-in process class that returns the process handle

```
initial
//set a seed at the start
process::self.srandom(100);
a = randomize(x);
...
end
```

You can spawn multiple concurrent threads using fork...join Each thread maintains its own random number generator: Seeded from the parent and can be manually reseeded

```
integer a, x, y, z;
...
fork

begin // use thread's initial seed
    #($urandom_range(2,0)) a = randomize(x);
end

begin // set seed at the start of thread
    process::self.srandom(100);
    #($urandom_range(2,0)) a = randomize(y);
end

begin // use initial seed then set new seed
    #($urandom_range(2,0)) a = randomize(z);
    process::self.srandom(200);
    #($urandom_range(2,0)) a = randomize(z);
end

// ADD MORE THREADS HERE -- NOT ABOVE!

join
```

This example illustrates two important features: Hierarchical seeding – the RNG of each forked thread is seeded from the next random value of the parent’s thread RNG. Thread stability – the sequence of values for x, y and z are dependent upon their order in the hierarchy, but not dependent on the order of process execution.

- 5) Use the **with** clause to attach a constraint block {...} containing: Constraint expressions or constraint expression ordering. Constraint expressions use two constructs for conditional constraints:

- implication: expression -> constraint_set
- if (expression) constraint_set [else constraint_set]

```
//type declarations as before
opcode_t  opcode;
regs_t    regs;
logic[7:0] data;
int ok;

// conditions: data only between 32 and 126
ok = randomize(data) with {data>=32; data<=126;};

// range: opcode in range of ADDI to SUBI or JMP to JMPC
ok = randomize(opcode) with { opcode inside { [ADDI:SUBI], [JMP:JMPC] } };

// distribution: REG0-REG1 twice as likely as REG2-REG3
ok = randomize(regs) with { regs dist { [REG0:REG1]:=2, [REG2:REG3]:=1 } };
```

```
logic[7:0] data;
...
// Randomize data with values < 100 when mode is small
// Randomize data with values > 200 when mode is large
// Don't constrain data if mode is not small or large
ok = randomize(data) with
    { mode == small -> data < 100;
      mode == large -> data > 200; };

// Payload can be constrained in the same way with if/else
ok = randomize(data) with
    { if (mode == small) data < 100; else
      if (mode == large) data > 200; };
```

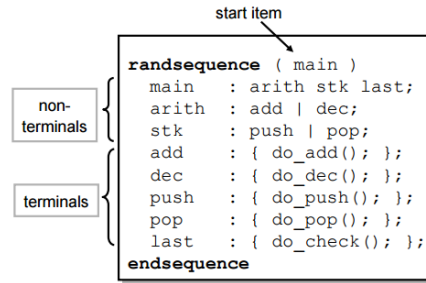
- 6) You can use the randcase keyword to randomly select a statement for execution. You can provide different probabilities, that is, weights, for each branch. The randcase weights are non-negative integral expressions that may be constant or variable. An item that you do not weight assumes the weight of 1. A weight of 0 indicates that the item shall never be selected.

```
for (int i=0; i<50; i++)
begin
    randcase
        20 : gen_atm;
        30 : gen_ethernet;
        10 : gen_ipv4;
        5 : gen_crc_error;
    endcase
```

```
for (int i=0; i<50; i++)
begin
    randcase
        a : gen_atm;
        a + b : gen_ethernet;
        a - b : gen_ipv4;
        b : gen_crc_error;
    endcase
```

Probability(gen_crc_error) = 5/65 ~ 8%

- 7) You can use randsequence keyword to execute a random sequence of statements based on rules called a production list. The argument to the randsequence construct is the identifier of the production to start with. This defaults to the first listed production. Each production consists of a function declaration, with optional parameter and return type declarations, followed by a colon, followed by rules for the production, where rules are separated by the bitwise OR (|) operator. Each rule is a production list of one or more productions, and may have a weight associated with it, and may have a code block associated with it. A code block is data declarations and statements between curly brackets ({}). Each randsequence construct creates its own automatic scope, which means that production declarations are not visible outside of the randsequence construct. Furthermore, each code block creates an anonymous automatic scope, so its declarations are also not visible outside the code block. The sequence in this example starts with the main production, which defines one production rule consisting of the arith, stk and last productions, in sequence. The arith production defines the add and dec alternative rules, each equally likely to be selected. The stk production defines the push and pop alternative rules, each also equally likely to be selected.



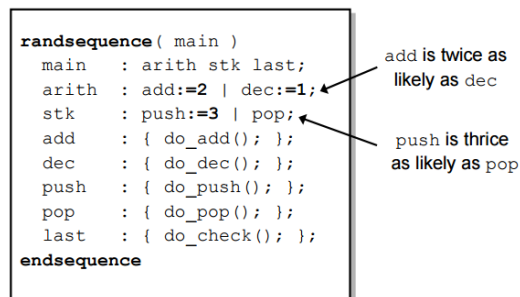
valid combinations

```

do_add() do_push() do_check()
do_dec() do_push() do_check()
do_add() do_pop()  do_check()
do_dec() do_pop()  do_check()

```

You can apply weights to production items to influence randomization



Note: every production declaration is a local function declaration, and every production item is a task call.

Productions can be conditional upon external variables. Use branching statements (case, if). You can use **break** and **return** in code blocks to abort productions.

- **Break:** terminates the entire randsequence production
- **Return:** aborts the generation of current production only

```

randsequence(test)
  test: run1 run2;
  run1: A B;
  run2: B C;
  A: { if(i==1)
      // exit item A
      return;
      $display("A"); };
  B: { if(i==2)
      // exit randsequence
      break;
      $display("B"); };
  C: { $display("C"); };
endsequence

```

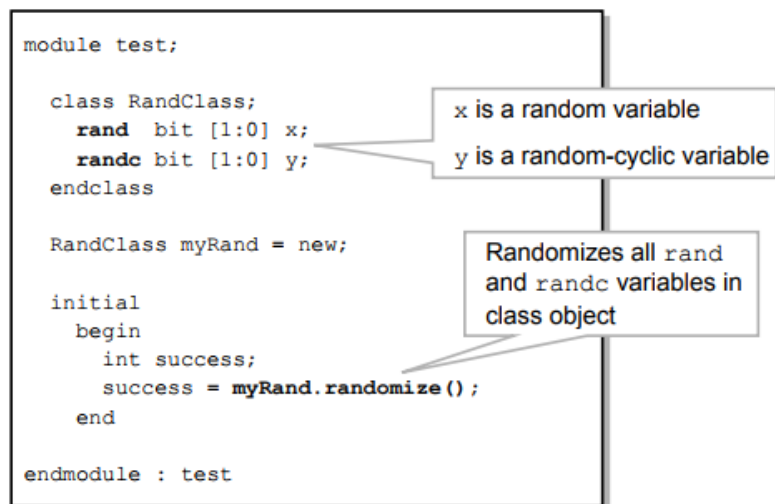
Chapter 7: class-based random stimulus

Topics

- 1) Randomization of class properties
- 2) Pre- and post-randomize methods
- 3) In-line control
- 4) Constraints
- 5) Setting seeds

- 1) Every class has a virtual member function called `randomize()` that returns an int value of 1 to indicate successful randomization, or 0 to indicate randomization failure. This function cannot be redeclared.

An example of this is:



- 2) The function `randomize()` calls automatically the member functions `pre_randomize()` and `post_randomize()`. These functions can be redeclared to perform any action before and after the randomization.
- 3) The function `rand_mode()` is used to find out if the randomization is enabled or not. The only argument that can be given to this function is a bit variable where 0 implies disabling and 1 implies enabling.
- 4) A constraint can be declared as a class member with the `constraint` keyword, followed by an identifier, followed by constraint block items enclosed within a set of curly (`{}`) braces. Constraint block items can be constraint expressions and can be statements ordering the constraint solution. Constraint expressions can be almost any integral expression. They are restricted to not have side effects, so you cannot use the assignment operators.

The inside operator is particularly useful in constraint expressions. An example of this is:

```
class RandClass;
  rand int x;
  randc bit [1:0] y;
  constraint c1 {x inside {3, 7, [11:20]};}
endclass

RandClass myRand = new;

initial
  begin
    int success;
    success = myRand.randomize();
    $display("x now ", myRand.x,,
            "y now ", myRand.y);
  end
```

c1 constrains x to the set 3, 7, 11-20

Moreover, weighted distributions and conditionals can be used inside constraint expressions. Besides, the constraint_mode() function is normally employed to manage constraints individually or all constraints of a class as follows:

```
class RandClass;
  rand bit [0:1] x;
  constraint blue {x != 2'b00;}
  constraint green {x != 2'b11;}
endclass

RandClass myRand = new;

initial
  begin
    int state, success;
    myRand.constraint_mode(0);
    myRand.blue.constraint_mode(1);
    state = myRand.green.constraint_mode();
    success = myRand.randomize();
  end
```

blue constrains x to not 00

green constrains x to not 11

disable all class object constraints

re-enable blue constraint

state of constraint green is still 0

a will be 01, 10, 11

- 5) The random() class method can be used to set manually the seed, by this way it can be forced to obtain different random values in each simulation.

```
class RandClass;
  rand int x;
  function new (int seed=5);
    this.random(seed);
  endfunction
endclass

RandClass myRand = new(7);
```