
Sem;Colon
Debug Your Soul

Session 3

Agenda

- More data types
- Scheduling Semantics
- Code coverage
- Functional coverage
- DO File
- Interface and Top Module

Prepared by :
Rana Ayman, Ammar Wahidi (Digital Team Semicolon)


More Data Types

Review of Arrays

```
reg [3:0] x ;
reg [0:3] y ;
initial

Begin
    x = 4'b1101 ;
    y = 4'b1101 ;
    $display(x[1:0]) ; → ?
    $display(y[1:0]) ; → ?
end
```

X

Bit [3]	Bit [2]	Bit [1]	Bit [0]

Y

Bit[0]	Bit[1]	Bit[2]	Bit[3]

Dynamic Array

- One dimension of an unpacked array whose size can be set or changed at run-time
- The space for a dynamic array doesn't exist until the array is explicitly created at run-time, space is allocated when new[Size] is called

```
// Declaration
bit [7:0] d_array1[ ];
int d_array2[ ];

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

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

Dynamic Array

```
// Change the length of the array after declaration
d_array1 = new[10];

// Allocate 6 new elements and retain values of 4
elements.
d_array1 = new[10](d_array1);

// Size of array (reduction method)
$display("Size of d_array1 %0d",d_array1.size());

// Delete array
d_array1. delete();
```

Queues

- Like a dynamic array, queues can grow and shrink
- Queue supports adding and removing elements anywhere

```
int    queue_1[$];     // queue of int
string queue_2[$];     // queue of strings

int K;

queue_1  = {0,1,2,3};
queue_2  = {"Red" ,"Blue" ,"Green"};

$display("Queue_1 size is %0d ",queue_1.size());
```

Queues

```
queue_2.insert(1,"Orange"); //{"Red" ,"orange" ,"Blue" ,"Green"}

queue_2.delete(3); //{"Red" ,"orange" ,"Blue“}

queue_1.push_front(22);//{22,0,1,2,3}

queue_1.push_back(44);//{22,0,1,2,3,44}

K = queue_1.pop_front();//{0,1,2,3,44} K =22

K = queue_1.pop_back();//{0,1,2,3} K =44

queue_1.delete();
```

Associative Arrays

int data[int] ;

- Memory is allocated only for stored elements
- Indexing is not limited to integers strings or enums can be used
- Ideal for sparsely populated data

```
int a_array [string]; // key is a string
int a_wild [*]; // Wild Card(any index type can take place, it can change)
 a_array["alpha"] = 10;
 a_array["beta"] = 20;
 a_array["gamma"] = 30;
// Here we assign using a string key to a_wild, so it becomes an
associative array with string keys
 a_wild [“string_wild"] = 100 ; // It is now in type string
 if (a_array.exists("beta"))
     $display("Key 'beta' exists with value: %0d", a_array["beta"]); // 20

 $display("Number of elements in a_array: %0d", a_array.num()); // 3
```

Associative Arrays

```
string s;
if (a_array.first(s))
    $display("First key: %s, Value: %0d", s, a_array[s]);//alpha 10

if (a_array.last(s))
    $display("Last key: %s, Value: %0d", s, a_array[s]);//gamma 30

s="beta";
if (a_array.next(s))
$display("next key: %s, Value: %0d", s, a_array[s]);//gamma 30

if (a_array.prev(s))
$display("prev key: %s, Value: %0d", s, a_array[s]);//beta 20

a_array.delete("beta");
a_array.delete();
```

Associative Arrays

Associative array methods

Method	Usage	Description
exists(i)	if (arr.exists(i))	Returns 1 if element with index i exists
num() / size()	arr.num()	Returns number of entries in the array
first(var)	arr.first(var)	Assigns to (var) the smallest index in the array
last(var)	arr.last(var)	Assigns to (var) the largest index in the array
next(var)	arr.next(var)	Assigns to (var) the next index after current (var).
prev(var)	arr.prev(var)	Assigns to (var) the previous index before current (var).

More data types

Array Manipulation Methods:

SystemVerilog provides several built-in methods to facilitate array searching, ordering, and reduction.

- Array Lector Methods
- Array Ordering Methods
- Array Reduction Methods

Array Manipulation Methods

Array Ordering methods

Method	Description
reverse()	reverses all the elements of the array
sort()	sorts the array in ascending order
rsort()	sorts the array in descending order
shuffle()	randomizes the order of the elements in the array

Array Manipulation Methods

Array Ordering methods

```
int my_array[5];
my_array = '{20, 10, 30, 50, 40};

my_array.sort();
$display("Array after sort():%p",my_array);//'{10, 20, 30, 40, 50}

my_array.reverse();
$display("\nArray after reverse() :%p",my_array);//'{50, 40, 30, 20, 10}

my_array.shuffle();
$display("\nArray after shuffle():%p",my_array);//'{10, 40, 50, 30, 20}

my_array.rsort();
$display("\nArray after rsort():%p",my_array);//'{50, 40, 30, 20, 10}
```

Array Manipulation Methods

Array Reduction methods

Method	Description
sum()	returns the sum of all the array elements
product()	returns the product of all the array elements
and()	returns the bit-wise AND (&) of all the array elements
or()	returns the bit-wise OR (|) of all the array elements
xor()	returns the logical XOR (^) of all the array elements

Array Manipulation Methods

Array Reduction methods

- The method returns a single value of the same type as the array element type without using With clause.
- The array elements must be of a type that supports bitwise operations (bits ,logic ,..)

```
        bit [3:0] my_array[];

initial begin
     my_array = new[5] ;
     my_array = '{2, 3, 5, 7, 11};
     $display("Sum: %0d", my_array.sum());//12
     $display("Sum: %0d", (my_array.sum() with(int '(item)) ));//28
     $display("Product: %0d", my_array.product());//6
     $display("Product: %0d", (my_array.product() with(int '(item)) ));//2310
     $display("Bitwise OR: %b", my_array.or());//1111
     $display("Bitwise XOR: %b", my_array.xor());//1000
     $display("Bitwise AND: %b", my_array.and());//0000
end
```

More data types

Real

- Floating Point 64 bit

Shortreal

- Floating Point 32 bit

Events

- An event is a static object handle to synchronize between two or more concurrently active processes. One process will trigger the event, and another process waits for the event.
- Can be assigned or compared to other event variables
  - Can be assigned to "**null**"
  - When assigned to another event, both variables point to same synchronization object
- Can be passed to queues, functions and tasks

```
event  over;                    // a new event is created called over
event  over_again = over;       // over_again becomes an alias to over
event  empty = null;            // event variable with no synchronization object
```

Events

```
ncsim> run
[0] Thread2: waiting for trigger
[0] Thread3: waiting for trigger
[20] Thread1: triggered event_a
[20] Thread2: received event_a trigger
[20] Thread3: received event_a trigger
ncsim: *W,RNQUIE: Simulation is complete
```

```
module tb;
  // Create an event variable that processes can use to trigger and wait
  event event_a;
  initial begin // Thread1: Triggers the event using "->" operator
    #20 ->event_a;
    $display ("[%0t] Thread1: triggered event_a", $time);
  end
  initial begin // Thread2: Waits for the event using "@" operator
    $display ("[%0t] Thread2: waiting for trigger ", $time);
    @(event_a);
    $display ("[%0t] Thread2: received event_a trigger ", $time);
  end
  initial begin // Thread3: Waits for the event using ".triggered"
    $display ("[%0t] Thread3: waiting for trigger ", $time);
    wait(event_a.triggered);
    $display ("[%0t] Thread3: received event_a trigger", $time);
  end
endmodule
```

Scheduling Semantics

Scheduling Semantics

Preponed: sample values before any update

Pre-Active (PLI): PLI callbacks before active region

Active: blocking assignments, continuous assignments, RHS evalution

Inactive: *#0* delays resume here

Pre-NBA / NBA / Post-NBA: non-blocking assignments processing

Pre-Observed: sampling before assertions check

Observed: concurrent assertions evaluated

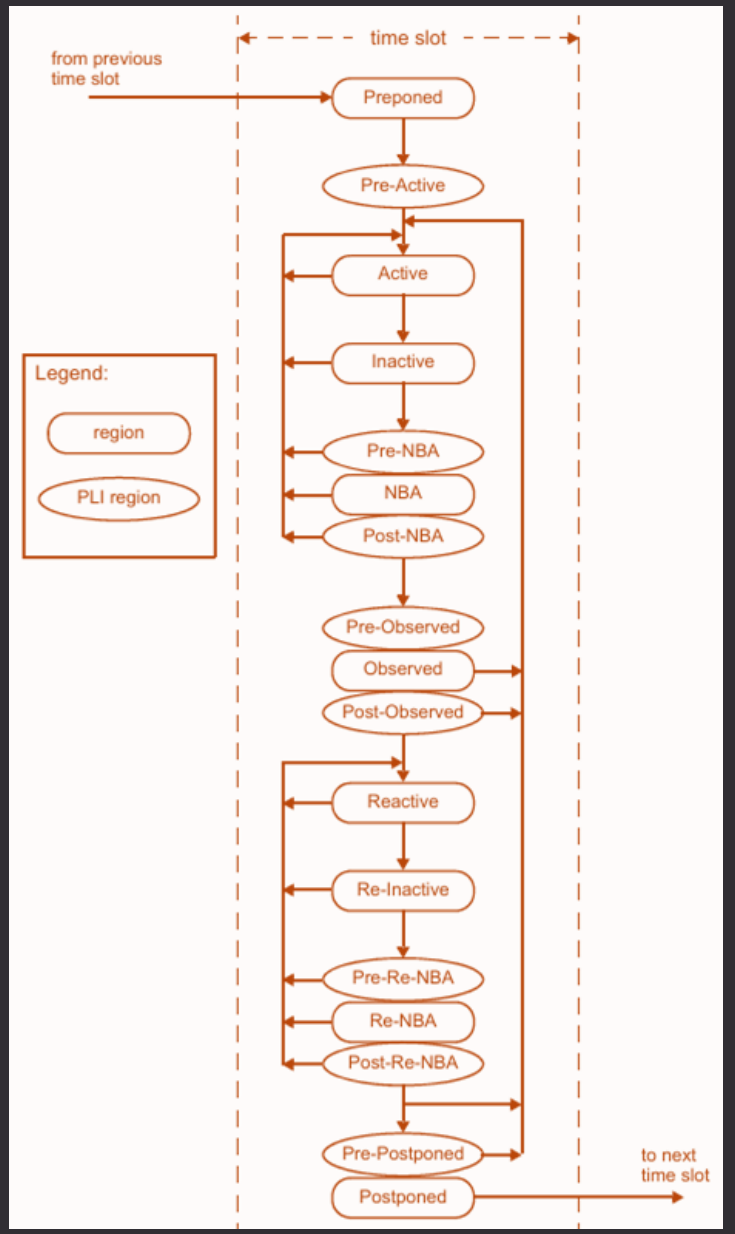
Post-Observed (PLI): PLI callbacks after assertion checks

Reactive: program blocks, testbench response

Re-Inactive, Pre-Re-NBA, Re-NBA, Post-Re-NBA: second iteration for reactive logic

Pre-Postponed: final PLI callback opportunity

Postponed: $monitor, $strobe, functional coverage



Scheduling Semantics

```
module sc_se();
  logic [7:0] a, b, c;
  initial begin
    a = 2;
    #2 c = 10;
    #4 b = 8;
  end
  initial begin
    a = 5; c = 6; b = 1;
    #2 c <= 8;
    #3;
    b <= 4;
    a = b;
    #1;
    #0 b = 15;
  end
endmodule
```

```
# T=0 :
  a=5
  c=6
  b=1
# T=2 :
  c=8 (NBA)
# T=5 :
  a=1
  b=4 (NBA)
# T=6 :
  b=15 (inactive)
```


Coverage

Code Coverage

- A metric used to determine how much of the design code is exercised during simulation.
- It helps ensure that all parts of the design (RTL) are tested, identifying untested areas
- Doesn’t measure the correctness of the design

```
module adder (
    input  clk,
    input  reset,
    input  signed [3:0] A,B, // Input data A and B in 2's complement
    output reg signed [4:0] C ); // Adder output in 2's complement
      always @(posedge clk or posedge reset) begin
      if (reset)
        C <= 5'b0;
      else
        C <= A + B;
   end
endmodule
```

Code Coverage

- Statement Coverage
- Branch Coverage
- Path Coverage
- Conditional Coverage
- Expression Coverage
- Toggle Coverage
- FSM Coverage

Code Coverage

Statement Coverage

- Verifies if each line of code (statement) is executed at least once

#Test: Reset = 1

```
 always @(posedge clk or posedge reset)
begin
   if (reset)
     C <= 5'b0;
   else
     C <= A + B;
end
```

¾ Coverage
75 %

Statement Coverage Report:

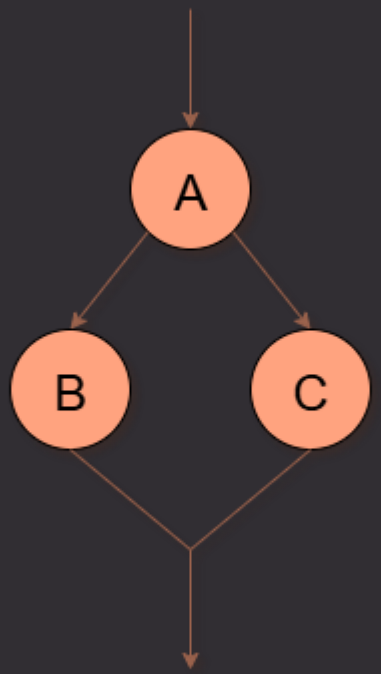
```
Statement Coverage:
    Enabled Coverage            Bins      Hits    Misses  Coverage
    ----------------            ----      ----    ------  --------
    Statements                     3         3         0   100.00%

================================Statement Details================================

Statement Coverage for instance /adder_tb2/a1 --

    Line         Item                     Count     Source
    ----         ----                     -----     ------
  File adder.v
    1                                               module adder (
    2                                                   input  clk,
    3                                                   input  reset,
    4                                                   input  signed [3:0] A, // Input data A in 2's complement
    5                                                   input  signed [3:0] B, // Input data B in 2's complement
    6                                                   output reg signed [4:0] C // Adder output in 2's complement
    7                                                           );
    8
    9                                                 // Register output C
    10           1                      11            always @(posedge clk or posedge reset) begin
    11                                                   if (reset)
    12           1                       2             C <= 5'b0;
    13                                                   else
    14           1                       9             C <= A + B;
```

Code Coverage

Branch Coverage

- Verifies if all branches (e.g., if ,case) have been evaluated to both true and false.

```
                          → A
always @(posedge clk or posedge reset)
   begin
      if (reset)
        C <= 5'b0;        → B
      else
        C <= A + B;       → C
   end
```



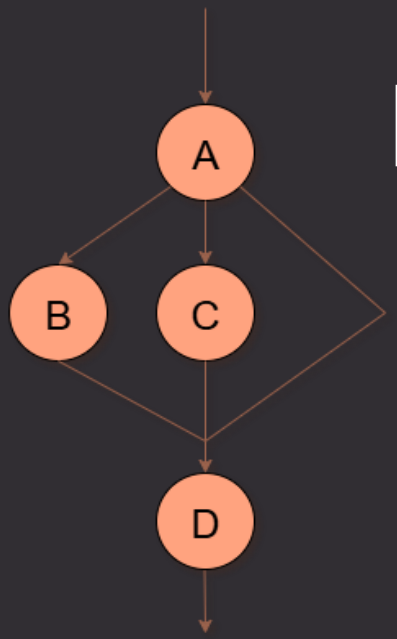
Code Coverage

Branch Coverage

- Verifies if all branches (e.g., if ,case) have been evaluated to both true and false.

Statement Coverage vs. Branch Coverage – How Do They Differ?

```
always @(*)
begin
    y = 5 ;                 → A
    if (load)               → B
    begin
        x = 5'b0;
    end
    else if (en)            → C
    begin
        x = A + B;
    end
    x = x + y ;             → D
End
```



3 Branches

Code Coverage

Branch Coverage

- Verifies if all branches (e.g., if ,case) have been evaluated to both true and false.

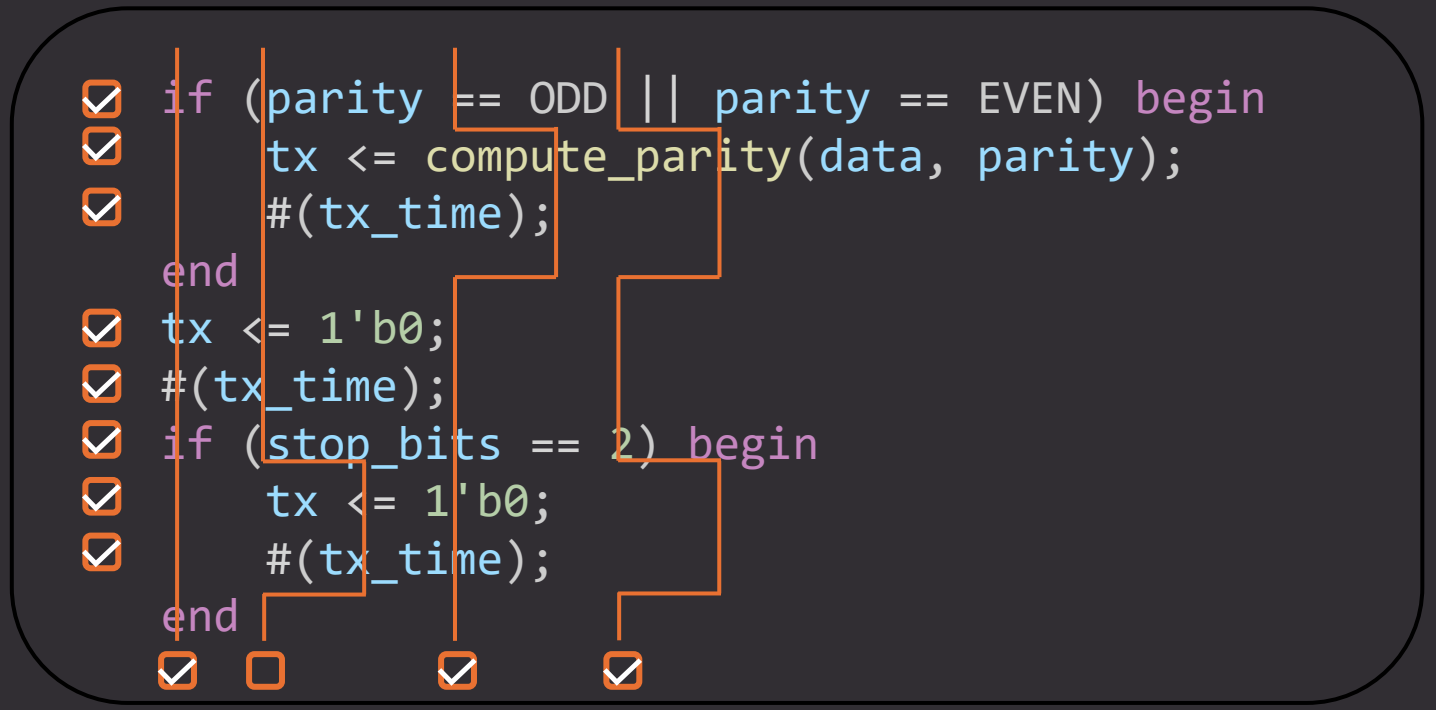
Branch Coverage Report:

```
----------------------------------------------------------------------------
Branch Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         2         2         0   100.00%

================================Branch Details================================

Branch Coverage for instance /adder_tb2/a1

    Line         Item                      Count     Source
    ----         ----                      -----     ------
  File adder.v
------------------------------------IF Branch------------------------------------
    11                                        11     Count coming in to IF
    11              1                          2           if (reset)
    13              1                          9           else
Branch totals: 2 hits of 2 branches = 100.00%
```

Code Coverage

Path Coverage

- Path coverage measures all possible ways you can execute a sequence of statements.



Code Coverage

Toggle Coverage

- Verifies if every bit of a signal toggles (0→1 and 1→0) during the simulation.

```
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                         30        29         1    96.66%

================================Toggle Details================================

Toggle Coverage for instance /adder_tb2/a1 --

                                          Node      1H->0L      0L->1H  "Coverage"
                                          ----------------------------------------
                                        A[0-3]           1           1      100.00
                                        B[0-3]           1           1      100.00
                                        C[4-0]           1           1      100.00
                                           clk           1           1      100.00
                                         reset           1           0       50.00

Total Node Count     =         15
Toggled Node Count   =         14
Untoggled Node Count =          1

Toggle Coverage      =      96.66% (29 of 30 bins)
```

Code Coverage

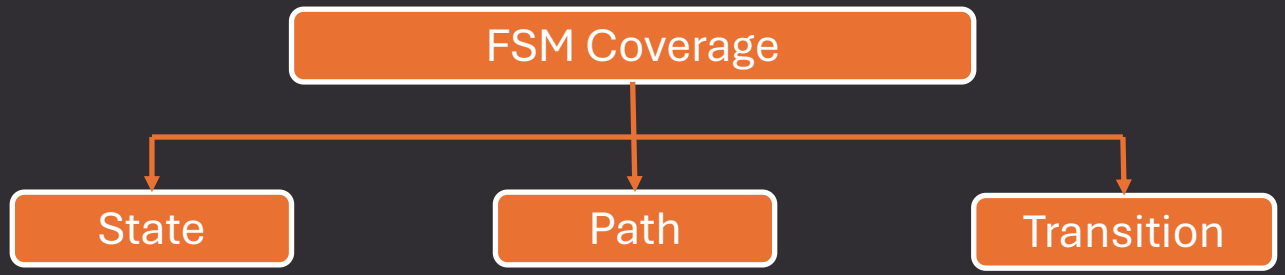
Code coverage

Four Possible outcomes:
True / True
True / False
False / True
False / False

Expression Coverage

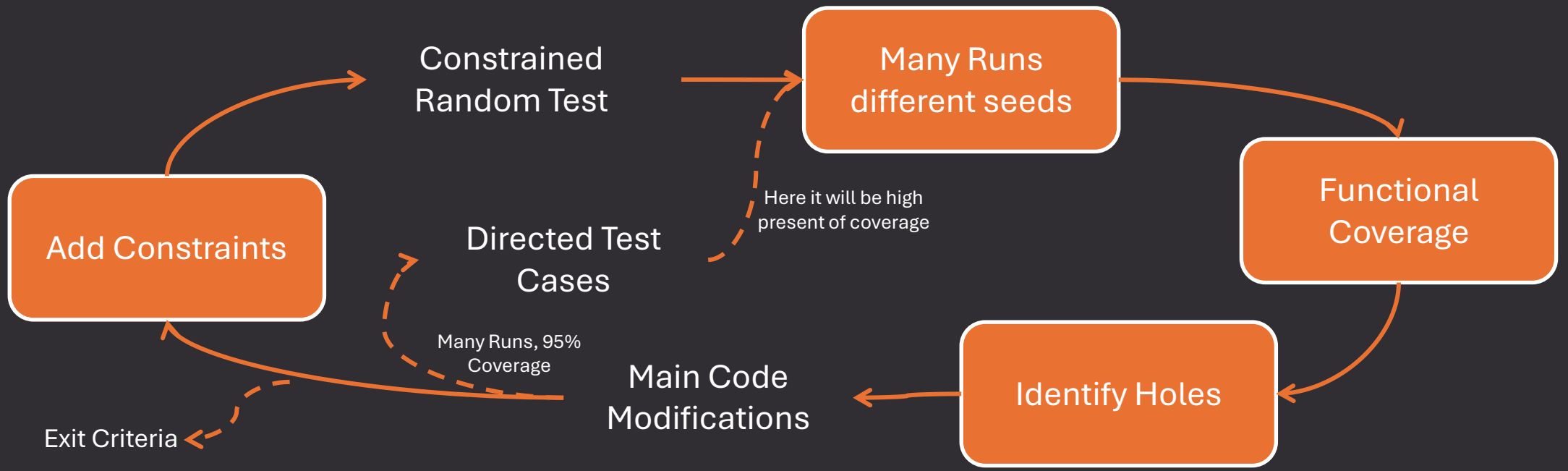
```
if ((y==1) || (x==0))
```

FSM Coverage



Functional Coverage

- Ensures that specific design features or scenarios are exercised during simulation
- It focuses on verifying that the **intended functionality** of the design is tested
- Starts from the design specification → used to build the verification plan and define coverage goals
- Coverage results are measured against exit criteria to decide when verification is complete



Functional Coverage

- Ensures that specific design features or scenarios are exercised during simulation
- It focuses on verifying that the **intended functionality** of the design is tested
- Starts from the design specification → used to build the verification plan and define coverage goals
- Coverage results are measured against exit criteria to decide when verification is complete

Functional coverage is user-defined and is implemented using:

1. Covergroups
2. Coverpoints
3. Bins
4. Cross Coverage

Functional Coverage

Covergroup

- The covergroup construct encapsulates the specification of a coverage model.
- It allows you to collectively sample variables that are sampled at the same clock (sampling) edge
- The type definition is written once, and multiple instances of that type can be created in different contexts.
- Can be defined in a module, interface, or class
- A covergroup instance created via the new() operator
- To trigger the coverpoint sampling you should use either clocking event or a sample task

```
covergroup cov_grp;
  cov_p1: coverpoint a;
endgroup

cov_grp cov_inst = new();
@(abc) cov_inst.sample();
```

```
covergroup cov_grp @(posedge clk);
  cov_p1: coverpoint a;
endgroup

cov_grp cov_inst = new();
```

Functional Coverage

Covergroup

Covergroup with arguments

- Covergroup can take optional arguments.

```
covergroup cg (ref int var, input bit mode);
  coverpoint var;
endgroup
```

- Providing arguments when creating an instance:
  - When you create (instantiate) a covergroup using new, you must pass in the actual values/variables for its arguments (unless defaults are given).
  - Ex:

```
cg mycg = new(my_var, 1'b0); // pass arguments in module
```

- Data types are integral only.
- Ref → samples live variable values (different instances can track different variables).
- Input → snapshot at creation (does not track updates) and not used. (default direction)
- Output/Inout → not allowed
- Arguments are read-only inside the covergroup.

Functional Coverage

Covergroup

Covergroup in Class

- When declared inside a class, a covergroup automatically creates an implicit instance with the same name.
- You need to call new for the covergroup inside the **class constructor**.
- If the covergroup has arguments:
  - The new call in the constructor must pass those arguments
  - Therefore, the class constructor must also take the same arguments

Functional Coverage

Covergroup

Covergroup in Class

```
bit clk ;
class xyz;
bit [3:0] m_x;
bit [3:0] m_y;
covergroup cov1 @(posedge clk) ;
// embedded covergroup, Type == instance
   coverpoint m_x;
   coverpoint m_y;
endgroup
function new();
    cov1 = new; // Covergroup handle
endfunction
endclass
```

```
module top();
xyz c_xyz ; // Instance
initial clk = 0;
always #5 clk = ~clk ;
always @(posedge clk) c_xyz.m_x ++;
always @(posedge clk) c_xyz.m_y ++;
initial
begin #12;
    c_xyz = new(); // Handle
end
endmodule
```

Functional Coverage

Coverpoint

Purpose of **Coverpoints:**

1. **Track Specific Values:**

   Ensure all important values of a signal are observed in your testbench.

2. **Analyze Ranges or Patterns:**

   Check if signals fall into predefined ranges or meet specific conditions.

3. **Enable Detailed Coverage Reporting:**

   Coverpoints provide metrics to identify tested and untested scenarios

Functional Coverage

Bins

bins are used to categorize or group the values of a coverpoint. They allow us to track how many times specific values, ranges, or patterns occur during simulation.

Types of Bins:

Automatic Bins:

- EDA Tool automatically creates bins for every unique value of the signal being covered

Explicit Bins:

- You manually define bins to specify values or ranges of interest.

Ignore Bins:

- Used to exclude certain values from coverage

Illegal Bins:

- Used to track invalid or illegal values that should never occur.

Functional Coverage

```
logic [7:0] addr;
  logic       wr_rd;

  covergroup cg @(posedge clk);
    c1: coverpoint addr
{ bins b1 = {0,2,7}; //increments for addr = 0,2 or 7

  bins b2[3] = {[11:20]};
//creates three bins b2[0],b2[1] and b2[3] distributed as follows:
(11,12,13),(14,15,16) and (17,18,19,20)

  ignore_bins b3   = {[30:40],[50:60],77};
//bin for addr = 30-40 or 50-60 or 77 values to exclude from functional coverage
calculation
```

Functional Coverage

```
  bins b4[] = {[79:99],[110:130],140}; //creates 43 bins

  ignore_bins b5[] = {1,2};  // These values will NOT be counted in coverage

 (hits are not required).
  illegal_bins b6[] = {160,170,180}; //creates three bins b5[0],b5[1] and b5[3]
with values 160,170 and 180 cause run time error
bins b7[] = {150,162,185,166};
bins b8[] = {150,170,185,198};//Overlapping Example
  bins b9    = {[200:$]}; //increments for addr = 200 to max value i.e, 255.

  bins b10[] = default;} // catches the values of the coverage point that do not
lie within any of the defined bins.
    c2: coverpoint wr_rd; //2 bins created automatically
  endgroup : cg
```

Functional Coverage

Transition Coverage

- The transition of coverage point can be covered by specifying the sequence
- It represents transition of coverage point value from value1 to value2

```
covergroup cg @(posedge clk);
  c1: coverpoint addr
  { bins b1   = (10=>20=>30);// transition from 10->20->30
    bins b2[] = (40=>50),(80=>90=>100=>120); // b2[0] = 40->50 and
b2[1] = 80->90->100->120
    bins b3 = (1,5 => 6, 7); // b3 = 1=>6 or 1=>7 or 5=>6 or 5=>7
    bins b4 = (1=>6) ,(1=>7) ,(5=>6) ,(5=>7);
    bins b5 = (0=>1[*2]=>2); //0=>1=>1=>2
    bins b6[] = (0=>1[->2]=>2);//0=>...1=>...1=>2
    bins b7[] = (0=>1[=2]=>2);//0=>...1=>...1=>...2
     }
endgroup : cg
```

Functional Coverage

Cross Coverage

Cross coverage in SystemVerilog is used to track **combinations of multiple coverpoints** to ensure that all important interactions between variables are tested. It allows us to verify that specific pairs (or groups) of conditions occur together.

Why Do We Need Cross Coverage?

- When writing functional coverage, we often define **separate coverpoints** for different signals. However, in real-world designs, multiple signals interact, and we need to check if all possible **combinations** of values appear in simulation.

Imagine verifying an **ALU (Arithmetic Logic Unit)**. We may need to check:

- Opcode
- Operand1
- Operand2

Simply covering these signals **individually** does not ensure that all valid **(opcode, operand1, operand2)** combinations appear in the simulation. This is where **cross coverage** helps.

Functional Coverage

Cross Coverage

```
covergroup cg ;
    cp_in :coverpoint seq_item.in ;
    cp_direction :coverpoint seq_item.direction ;
    cross cp_in ,cp_direction;
    endgroup
```

```
----------------------------------------------------------------------------------------------------
TYPE /shifter_coverage_pkg/shifter_coverage/cg       100.00%        100          -    Covered
    covered/total bins:                                    8          8          -
    missing/total bins:                                    0          8          -
    % Hit:                                           100.00%        100          -
    Coverpoint cp_in                                 100.00%        100          -    Covered
        covered/total bins:                                2          2          -
        missing/total bins:                                0          2          -
        % Hit:                                       100.00%        100          -
        bin auto[0]                                       51          1          -    Covered
        bin auto[1]                                       49          1          -    Covered
    Coverpoint cp_direction                          100.00%        100          -    Covered
        covered/total bins:                                2          2          -
        missing/total bins:                                0          2          -
        % Hit:                                       100.00%        100          -
        bin auto[RIGHT]                                   36          1          -    Covered
        bin auto[LEFT]                                    65          1          -    Covered
    Cross #cross__0#                                 100.00%        100          -    Covered
        covered/total bins:                                4          4          -
        missing/total bins:                                0          4          -
        % Hit:                                       100.00%        100          -
        Auto, Default and User Defined Bins:
            bin <auto[1],auto[LEFT]>                      31          1          -    Covered
            bin <auto[0],auto[LEFT]>                      34          1          -    Covered
            bin <auto[1],auto[RIGHT]>                     18          1          -    Covered
            bin <auto[0],auto[RIGHT]>                     17          1          -    Covered
Statement Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                       4         4         0   100.00%
```

Functional Coverage

Cross coverage

Exclude bins (With, bins of and intersect)

```
covergroup cg @(posedge clk);
  coverpoint opcode {
    bins add = {ADD};
    bins sub = {SUB};
    bins mul = {MUL};
    bins div = {DIV};}
  coverpoint operand {
    bins max  = {127};
    bins min  = {-128};
    bins zero = {0};
    bins other = default;}
  cross opcode, operand {
    ignore_bins div_by_zero = binsof(opcode.div) && binsof(operand.zero);
    ignore_bins div_by_zero = binsof(opcode) intersect {DIV} && binsof(operand)
intersect {0}; }
endgroup
```

Functional Coverage

Cross coverage

Exclude bins (With, bins of and intersect)

```
bit [2:0] a,b,c ;
covergroup cov @(posedge clk);
coverpoint a {
    bins bin_1 = {1,2,3};
    bins bin_2 = {4,5,6,7};}
coverpoint b {
    bins bin_3 = {1,2,3};
    bins bin_4 = {4,5,6,7};}
c_w:coverpoint c {
    bins bin_w[] = c_w with (item >2); // {3,4,5,6,7}}
cross a,b {
    bins bin_5 = binsof(a.bin_2);
    bins bin_6 = binsof(a.bin_1 || b.bin_3);
    bins bin_7 = binsof(a.bin_1 && b.bin_3); }
endgroup
```

Functional Coverage

Coverage Options

Option	Description
option.weight	Assigns weight to a coverpoint or cross. Default is **1**.
option.auto_bin_max	Specifies the max number of automatically created bins. Default is **64**.
option.comment	Adds a string comment to a coverpoint or cross coverage. Default is **“”**
option.at_least	Specifies the min number of times a bin must be hit to be considered covered. Default is **1**.
option.cross_auto_bin_max	Controls the max number of bins for **cross coverage**.
option.goal	Sets the percentage target for coverage completion. Default is **100**.

Functional Coverage

Coverage Options

```
cross opcode , operand
  {
    bins op = binsof (opcode.add) && (binsof (operand.max) || binsof(operand.min));
    option.weight = 5;//it means the bin will be counted 5 times each time it is hit.
    option.cross_auto_bin_max =0;
  }
```

Functional Coverage

Coverage Methods

Methos (functions)	Description	Can be called on
void sample	Triggers covergroup sampling	covergroup
void start	Starts collecting coverage information	covergroup, coverpoint, cross
void stop	Stops collecting coverage information	covergroup, coverpoint, cross
void set_inst_name	Sets instance name to the given string	covergroup
real get_coverage	Returns cumulative or type coverage of all instances of coverage item.	covergroup, coverpoint, cross
real get_inst_coverage	Returns specific instance coverage on which it is called.	covergroup, coverpoint, cross

Functional Coverage

Coverage Methods

```
module func_coverage;
  bit [7:0] addr, data;
  covergroup c_group;
    cp1: coverpoint addr;
    cp2: coverpoint data;
    cp1_X_cp2: cross cp1, cp2;
  endgroup : c_group
  c_group cg = new();

  initial begin
    cg.start();
    cg.set_inst_name("my_cg");
    forever begin
      cg.sample();
      #5;
    end
  end
```

```
  initial begin
    $monitor("At time = %0t: addr = %0d,
data = %0d", $time, addr, data);
    repeat(5) begin
      addr = $random;
      data = $random;
      #5;
    end
    cg.stop();
    $display("Coverage = %f",
cg.get_coverage());
    $finish;
  end

endmodule
```

```
At time = 0: addr = 36, data = 129
At time = 5: addr = 9, data = 99
At time = 10: addr = 13, data = 141
At time = 15: addr = 101, data = 18
At time = 20: addr = 1, data = 13
Coverage = 5.777995
```

DO File

Do File

```
vlib work
vlog ALSU_pkg.sv ALSU.v ALSU_tb.sv +cover
vsim -voptargs=+acc work.ALSU_tb -cover
add wave *
coverage save ALSU_test.ucdb -onexit
run -all
vcover report ALSU_test.ucdb -details -all -annotate -output ALSU_cvr.txt
quit -sim
```

- `vlib work` — Creates work library
- `vlog ALSU_pkg.sv ALSU.v ALSU_tb.sv +cover` — Compile files and enables coverage tracking
- `vsim -voptargs=+acc work.ALSU_tb -cover` — Simulate testbench and ensure coverage metrics are tracked
- `add wave *` — Add all waves in the testbench
- `coverage save ALSU_test.ucdb -onexit` — Saves the collected coverage data into a **Unified Coverage Database (UCDB)** file
- `run -all` — Run the simulation
- `vcover report ALSU_test.ucdb -details -all -annotate -output ALSU_cvr.txt` — Writes the coverage report
- `quit -sim` — End the simulation

Interface

Interface

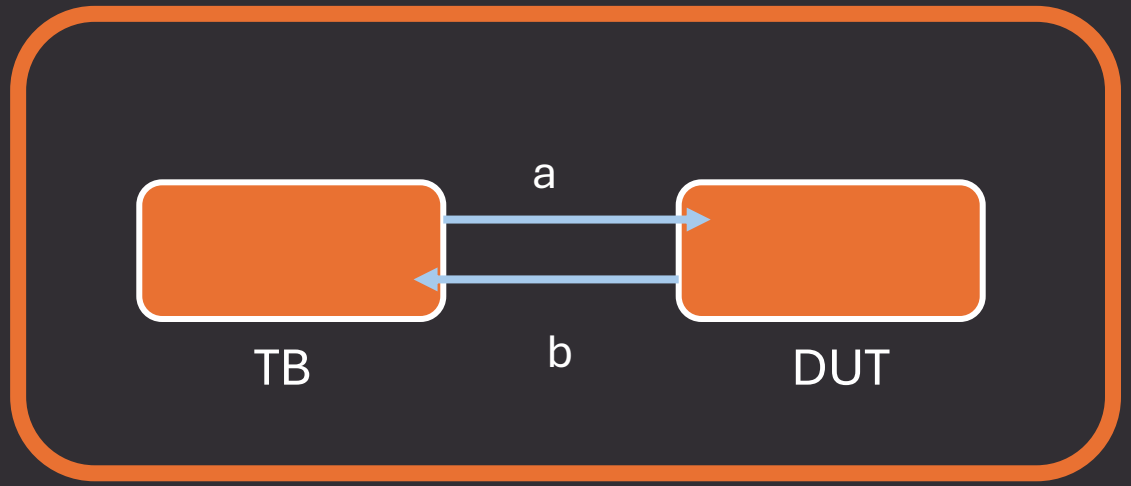
It is a mechanism used to group signals and simplify connectivity between modules. It is widely used in **verification** and **design abstraction**

Why Use Interfaces?

- **Reduces Code Complexity** → Instead of passing multiple signals separately, we pass a **single interface**.
- **Improves Readability & Maintainability** → Changes in signals only require modifications inside the interface, not across multiple modules.
- **Supports Modports** → Restricts signal directions for different module roles (e.g., Master/Slave).
- **Easy Reusability** → Can be reused across multiple designs, making testbenches and RTL modular.

Interface

```
module top ();
logic [3:0] a , b ;
dut DUT (a,b) ;
tb TB (a,b) ;
endmodule
```

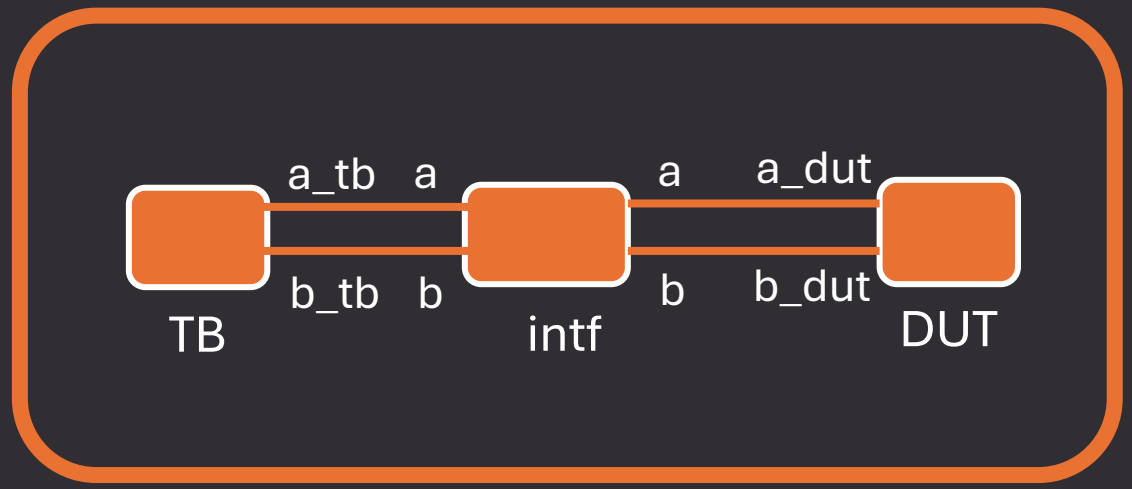



```
module tb (
input   logic [3:0] b_tb ,
output  logic [3:0] a_tb
);
endmodule
```

```
module dut (
input   logic [3:0] a_dut ,
output  logic [3:0] b_dut
);
endmodule
```

Interface

```
module top ();
intf intf1 ();
dut DUT (intf1.a,intf1.b) ;
tb TB (intf1.a,intf1.b) ;
endmodule
```



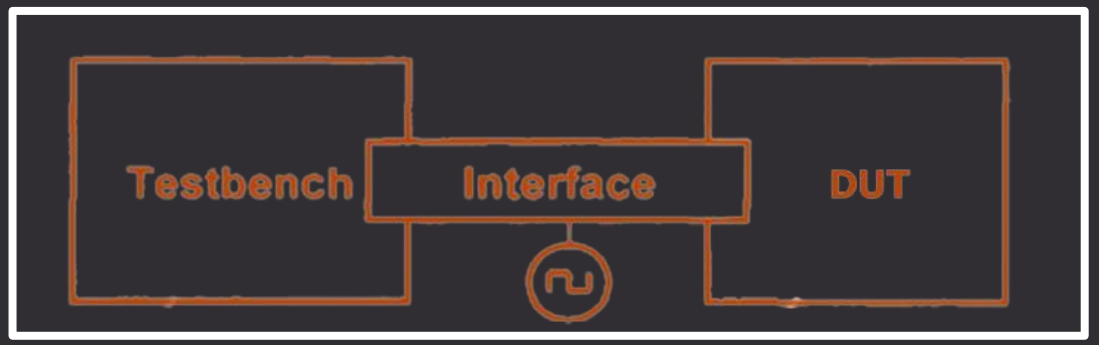

```
interface intf;
    wire [3:0] a ;
    wire [3:0] b ;
    // a and b here are inout
direction (as default)
endinterface
```

```
module dut (
input   logic [3:0] a_dut ,
output  logic [3:0] b_dut
);
endmodule
```

```
module tb (
input   logic [3:0] b_tb ,
output  logic [3:0] a_tb
);
endmodule
```

Interface

Modport




```
module top ();
intf intf1 ();
dut DUT (intf1) ;
tb TB (intf1) ;
endmodule
```

```
module dut (intf.dut intf1);
logic [3:0] a_dut ;
logic [3:0] b_dut ;
assign a_dut = intf1.a ;
assign intf1.b = b_dut ;
endmodule
```

```
interface intf;
    wire [3:0] a ;
    wire [3:0] b ;
    modport dut (input a, output b);
    modport test (input b, output a);
endinterface
```

```
module tb (intf.test intf1);
logic [3:0] b_tb  ;
logic [3:0] a_tb  ;
assign b_tb = intf1.b ;
assign intf1.a = a_dut ;
endmodule
```

Improving Our Testbench

Improving our testbench

```
module adder (a, b, c ,clk);

input [3:0] a, b ;
input clk;
output reg [4:0] c;

always @(posedge clk) begin
        c = a + b;
end

endmodule
```

Improving our testbench

```
import adder_sequence_item_pkg::*;
import adder_monitor_pkg::*;
module adder_tb();
logic [3:0] a ,b;
logic [4:0] c;
bit clk;
adder_sequence_item_class adder_tb_sequence_item_object = new();
adder_monitor_class adder_tb_monitor_object = new();
adder DUT (a,b,c,clk);
initial begin
    clk = 0;
    forever begin
        #20 clk = ~clk;
    end
end
```

Improving our testbench

```
initial begin
    for (int i = 0 ; i<100 ;i++ ) begin
        assert (adder_tb_sequence_item_object.randomize());
        a = adder_tb_sequence_item_object.a;
        b = adder_tb_sequence_item_object.b;
        @(negedge clk);
        adder_tb_sequence_item_object.c = c;
        adder_tb_monitor_object.monitor(adder_tb_sequence_item_object);
        end
        @(negedge clk);
        $stop;
end
endmodule
```

Improving our testbench

```
interface adder_if (clk);
input bit clk ;


logic [3:0] a,b ;
logic [4:0] c ;


modport DUT (input a,b,clk, output c);
modport Test (input c,clk, output a,b);


endinterface : adder_if // Interface
```

```
module top ();
bit clk ;
always #1 clk = ~clk ;
adder_if iff (clk);
adder dut (iff);
adder_tb tb (iff);
endmodule : adder_top // TOP
```

Improving our testbench

```
module adder (adder_if.DUT Iff);
always @(posedge Iff.clk) begin
Iff.c = Iff.a + Iff.b;
end
endmodule : adder // DUT
```

```
import adder_sequence_item_pkg::*;
import adder_monitor_pkg::*;
module adder_tb(adder_if.TEST Iff);
adder_sequence_item_class adder_tb_sequence_item_object = new();
adder_monitor_class adder_tb_monitor_object = new();
initial begin
for (int i = 0 ; i<1000 ;i++ ) begin
    assert (adder_tb_sequence_item_object.randomize());
    Iff.a = adder_tb_sequence_item_object.a;
    Iff.b = adder_tb_sequence_item_object.b;
    @(negedge Iff.clk);
    adder_tb_sequence_item_object.c = Iff.c;
    adder_tb_monitor_object.monitor(adder_tb_sequence_item_object);
end
@(negedge Iff.clk);
$stop;
end
endmodule : adder_tb // TB
```

Improving our testbench

```
package adder_sequence_item_pkg;

    class adder_sequence_item_class;

        rand logic [3:0] a, b;
        logic [4:0] c;
        bit clk;

        constraint constraint_1 {
            a dist {[0:14]:/70,15:/30};
            b dist {[0:14]:/70,15:/30};
        }

    endclass
endpackage
```

Improving our testbench

```
package adder_coverage_pkg;
    import adder_sequence_item_pkg::*;
    class adder_coverage;
     adder_sequence_item_class seq_item = new();
    covergroup cg ;
    cp_a :coverpoint seq_item.a {
        bins max = {15};
        bins zero = {0};
        bins other = default;
    }
    cp_b :coverpoint seq_item.b ;
    cp_c :coverpoint seq_item.c {
        bins max = {30};
        bins min = {0};
        bins other = default;
    }
cross cp_a ,cp_b ;
    endgroup
```

```
  function sample_data
(adder_sequence_item_class item);
        seq_item = item;
        cg.sample();
    endfunction

    function new ();
            cg=new();
    endfunction

    endclass : adder_coverage

endpackage : adder_coverage_pkg
```

Improving our testbench

```
package adder_monitor_pkg;

    import adder_sequence_item_pkg::*;
    import adder_scoreboard::*;
    import adder_coverage_pkg::*;

    class adder_monitor_class;

        adder_scoreboard_class monitor_scoreboard_object = new();
        adder_coverage cvr = new();


            task monitor(adder_sequence_item_class value_extraction_object);
                    monitor_scoreboard_object.scoreboard(value_extraction_object);
                    cvr.sample_data(value_extraction_object);
            endtask
    endclass
endpackage
```

Improving our testbench

Cross #cross__0#	93.75%	100	-	Uncovered
covered/total bins:	30	32	-	
missing/total bins:	2	32	-	
% Hit:	93.75%	100	-	
Auto, Default and User Defined Bins:				
bin <zero,auto[15]>	20	1	-	Covered
bin <zero,auto[14]>	1	1	-	Covered
bin <zero,auto[13]>	3	1	-	Covered
bin <zero,auto[12]>	5	1	-	Covered
bin <zero,auto[10]>	3	1	-	Covered
bin <zero,auto[9]>	6	1	-	Covered
bin <zero,auto[8]>	4	1	-	Covered
bin <zero,auto[7]>	5	1	-	Covered
bin <zero,auto[6]>	2	1	-	Covered
bin <zero,auto[5]>	7	1	-	Covered
bin <zero,auto[3]>	4	1	-	Covered
bin <zero,auto[2]>	2	1	-	Covered
bin <zero,auto[1]>	4	1	-	Covered
bin <zero,auto[0]>	2	1	-	Covered
bin <max,auto[15]>	84	1	-	Covered
bin <max,auto[14]>	18	1	-	Covered
bin <max,auto[13]>	10	1	-	Covered
bin <max,auto[12]>	11	1	-	Covered
bin <max,auto[11]>	11	1	-	Covered
bin <max,auto[10]>	8	1	-	Covered
bin <max,auto[9]>	10	1	-	Covered
bin <max,auto[8]>	11	1	-	Covered
bin <max,auto[7]>	12	1	-	Covered
bin <max,auto[6]>	15	1	-	Covered
bin <max,auto[5]>	11	1	-	Covered
bin <max,auto[4]>	10	1	-	Covered
bin <max,auto[3]>	18	1	-	Covered
bin <max,auto[2]>	16	1	-	Covered
bin <max,auto[1]>	17	1	-	Covered
bin <max,auto[0]>	18	1	-	Covered
bin <zero,auto[11]>	0	1	1	ZERO
bin <zero,auto[4]>	0	1	1	ZERO

Classwork

```
module counter(clk , load , up , down ,rst, in_value, counter , high , low);
input clk, load, up, down, rst;
input [4:0] in_value;
output high, low;
output reg [4:0] counter;
always @(posedge clk or posedge rst) begin
    if(rst)
        counter <= 0;
    else begin
    if(load)
        counter <= in_value;
    else if(down && !low)
        counter <= counter - 1'b1;
    else if(up && !high)
        counter <= counter + 1'b1;
    end
end
assign high = (counter == 5'b11111) ? 1 : 0;
assign low = (counter == 5'b00000) ? 1 : 0;
endmodule
```

Thank You

References

- SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Chris Spear , Greg Tumbush, 2012
- Writing Testbenches using SystemVerilog , Janick Bergeron Synopsys, Inc. 2006
- IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language
- Questa® SIM Command Reference Manual Including Support for Questa Base
- ChipVerify
- Home - VLSI Verify
- https://theartofverification.com/types-of-coverage-metrics/?utm_source=chatgpt.com
- https://uobdv.github.io/Design-Verification/Supplementary/An_Introduction_to_FSM_Path_Coverage.pdf?utm_source=chatgpt.com
- https://vlsiworlds.com/system-verilog/coverage-methods-in-systemverilog/