19-11-2024

System Verilog

-Ashlesha Gokhale

Introduction, history, overview.

↓

Property Specific Langu

Super log -> as a standard

-> 1800-2013 latest.

Key features:

Design - Module, interface, hierarchi.

Verification - program, constrain, coverage

Data type - logic, bit, enum, struct...

Object oriented - class, semaphore...

Property specification - Assertion

Preponed
↓
Active
↓
Inactive
↓
NBA
↓
Observed
↓
Reactive
↓
Postponed

Declaration spaces

-Module, Program, Interface,

↓ Continous procedural generate specify

↓ Initial

x continue assign always generate specify

↘ data deleration
Subroutine definition
interface inclu
process (initial not reconeded)
mod ports -> to give diraction to signal inside interface.
clocking block
↘ clocking → ↘ Active
end clocking. ↳ Reactive region ✓

↓

Simulation S
-> change event

↓

Simulation should excite in Reactive regin

Tx Rx
x, z propagation
check

Checker

- to ~~validate~~ provide assertions to validate design.

Package

- sharing parameter, data, task, function, sequence, checker
- not implicit continious assignment
- process not allowed directly

Configuration(VHDL)

- to bind different sowce to particular instance of design

```
config cfg1;
  design rtlLib.top;
  default liblist rtlLib;
  instance top.a2 liblist gateLib;
endconfig
```

```
>> vlogan -work rtlLib -sverilog ./rtlLib/*.sv
>> vlogan -work gateLib -sverilog ./gatelevel/*.sv

>> vcs -full64 -top top -lib rtlLib -lib gateLib
```

Package.

Program

Interface

Modport

Process

Clocking

Checker

Data Types

4 State data types → integer, logic, reg, time } → 2 bit

2 State data types → bit, byte, int, short int, long int

- 2 State → non-Synthesizable
- bit → 1 bit
- byte → 8 bit
- int → 32 bit
- short int → 16 bit
- long int → 64 bit

- String a = " ... "
- String b = { " ... ", a } → concatenation
- { 16 { "0" } } → replication
- a[3] → indexing.

- .len()
- .putc(i, c)
- .getc(i)
- .toupper()
- .tolower()
- .compare()
- .icompare()
- ~~.substring~~ .substr(i, j) — i^{th} to j^{th} position

atoi
atohex
atooct
atobin
atoreal
itoa ← void fn => str.itoa(i);
hextoa
octoa
realtoa

- event [->] have persistant triggered state that lasts for duration of step.
  - [->] : operator

```
event done, blast;
event done_too = done;

task trigger (event ev);
    -> ev;
endtask

fork
  @done_too;                  // wait for done through done too
  #1 trigger(done);           // trigger done through task trigger
join

fork
  -> blast;
  wait(blast.triggered);
join
```

triggered : every event have

o → userdefined type : type- def. typedef ↳ as vector multiple to change datatype

o → enum → first - first element
last - last element.
name - name of element
next - next element } wrap around
previous - previous element
num - number of elements

o → type
var type c;
c = type A.bus;

- Type casting. int'(r * 3.14);
- Size casting 16'(a+5); → to make intended truncation.
- Sign casting signed'(a);

$ cast → as a task (or) function

Operator

>>> → rotate, arithmatic shift.

==? → wild card → x, z (as) don't care at matching.
!=?

inside. → if (data inside {[0:255]})
if (data inside {3'b1?1})
if (data inside {myarray})

21-11-2024

Aggregate Data Types

- multi dimentional array.
- structure → anonymous ⇒ two anonymous structure with identical data types inside
  → named. treated different by compiler emp1 = emp2
  ⇒ Not possible
  → to be type casted.
  - packed
  - unpacked (default)
  - all field are allotted consequtively
- union → packed (only some size) either vector or scalar)
  → unpacked (same size → only lower bits are used)
  → tagged, nested with
  → named
- array - packed, unpacked

  [ ][ ] mem [ ]

  packed → memory unpacked

  ⇒ '{default: 8'h00}; // All value initilized to 0 allotted consecutively

  ⇒ A[x+:c] , A=+3 .

  ↓ const

  → left side of multi-dimentional array never fail

  dynamic (non-synthesizable) → Depth FIFO
  corner case

  → associative array → content addrable memory CAM
  {key : value} - memory allocated only when element is intalized

  int <arrname> [string];

  [*]; → also permitted.

  ↳ real, array, class object (not allowed)

- queues - <type> <name> [$];

  pop_front, pop_back, push_front, push_back
  {6,9} {9,6}

  $ last element.

Array methods → find, find_first, min, max, unique [Manipulation]

reverse, sort, rsort, shuffle + "with" [ordering methods]

c.sort with (item.red);

sum, product, and, or, xor [reduction method]

22-11-2024

Procedural Block

- inial
- final
-
- always
- always_comb → @* sometime misses corner case
- always_latch
- always_ff

Decision statements

- if else
- case, casex, casez
- unique, unique0 ≠ priority ??? → given warnings as per

Loops

- for, for_each

  → for(int i=0; i<N; i++)

  ↳ not allowed in verilog ← multiple variable allowed declare, inc

- break ... continue
- fork join → join_none
  → join_any

  → disable fork, wait fork

}

Task & function

- automatic → opp. static (default) → functions inside class are automatic by default.
  - ↳ same memory used again & again
- function → input, output → not in Verilog
  - ↳ inout.
  - ↳ void
- ** argument as input by (default)
- default arguments defined

fn Ca < >, op < >)

23/11/2024

State machine for AHP address, data conflict

Ready & not IDLE

Idle → A_0D_x → A_1D_0 → A_0D_1

Ready not IDLE/Busy

Ready not IDLE

Ready & not IDLE → HTRANS

IDLE => Ready & not IDLE/Busy — A_0D_x

| IDLE — IDLE

A_0D_x → Ready not IDLE — A_1D_0

| IDLE — IDLE

| notready — A_1D_0

A_0 A_1 A_2 A_3 } without wait A_0 A_1 A_1 A_2 } with ready.

D_0 D_1 D_2 D_0 D_0 D_1

Ready 0 1

(IDLE) $A_x D_x$ → Ready ← SEQ | NONSEQ $A_0 D_x$

|

| IDLE | Busy IDLE

$A_0 D_x$ → Ready ← SEQ | NONseq $A_1 D_0$

| not Ready $A_0 D_x$

| IDLE | Busy ~~$A_0 D_x$~~ $A_x D_0$ (new data)

$A_1 D_0$ → Ready ← SEQ | NONSEQ $A_0 D_1$ ($A_2 D_1$)

($A_0 D_0$ ⇒ data write)

mem[A_0] <= D_0;

| not ready $A_1 D_0$

| IDLE | Busy ~~A_x IDLE~~ $A_x D_0$ new..

$A_0 D_1$ → Ready ← SEQ | NONSEQ $A_1 D_0$

mem[A_1] <= D_1;

| not ready $A_0 D_1$

| IDLE ~~IDLE~~ $A_x D_0$

$A_x D_1$ → Ready ← SEQ | NONSEQ $A_1 D_0$

| not Ready. $A_x D_1$

| IDLE $A_x D_0$

$A_x D_0$ → Ready ← SEQ | NONseq $A_0 D_1$

| not Ready $A_x D_0$

IDLE ~~IDLE~~ $A_x D_0$

(IDLE) A_x B_x → Ready & SEQ / NON seq → $A_0 D_x$

| → IDLE → IDLE

A_0 D_x → Ready & SEQ / NONSEQ → A_1 D_0

| → A_0 D_x

| not Ready → ~~IDLE~~ IDLE → A_x D_0

A_1 D_0 → Ready & SEQ / NONseq → A_0 D_1

| → A_1 D_0

| not ready → IDLE → A_x D_1

A_0 D_1 → Ready & SEQ / NON seq → A_1 D_0

| → A_0 D_1

| not ready → IDLE → A_x D_0

A_x D_1 → Ready & SEQ / NON seq → A_0 D_x

| → A_x D_1

| not ready → IDLE → A_x D_x (IDLE)

A_x D_0 → Ready & SEQ / NON seq → A_0 D_x

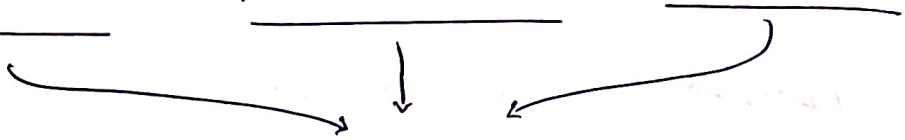
| → A_x D_0

| not ready → IDLE → A_x D_x (IDLE

25-11-2024

Verification → functional, timing, power.

Design intent circle, specification circle, implementation circle.



making all circle to coincide.

Bug can occur due to

- mis-interpreted specification
- misunderstanding of design
- incorrect specification
- missed scenarios
- assumption - (most cause for buggy code)

Cost of bug at different phase of verification

- Unit level verification — $
- Subsystem level verification — $$
- System level verification. — $$$
- Validation.
- Application
- Customer.

Types of bug

- Specification Bug
- Performance Bug
- Interface Bug

Parts of testbench

Generator, driver, monitor, checker.

Finding Bug

as bug could be deep within

activate → propagate → detect

Essential parts in testcase

Configuration, scenario generation, some checking function

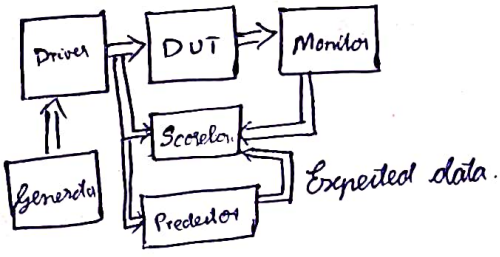
Task based testbench

- Task reset (no input required)
- Task init input i/p (long, short, bresh)
- Task gen_rcvd_ok_pulse i/p (~~none~~ pulse width)
- Task check_count i/p (expected count)
- Task check_intr i/p (expected val)

AHB Test case
↳ Driving data & address

Component based testbench.

Driver, generator, monitor, scoreboard. predictor



Types of verification.

→ Formal verification → Mathematical model of DUT compared according.

- model checking
- equivalence checking

→ Funtional verification.

26-11-2024

Object oriented programming.

→ objects are non-synthesizable

→ instanciate object only inside program ... endprogram

→ new ( ) [constructor] ← arg

- p1 = p2 ← change ref name [new address space not alloted]
- p1 = new p2 ← new p1 created with p2 values copied.
  ↳ shallow copy (objects within are not copied)

Subclasses (derived class)

→ extends

→ super.new(a);

→ this.i = i;

Data Encapsulation

local, protected → can be accessed within class & derived class.
↓
accessed only within class

↓
accessed using setter & getter methods.

Polymorphism

- virtual → in base class for overridden.
- calc(num) calc(num, num) → overloading → To try in lab both

no method overloading based on different return type

Abstract Class

- virtual → object cannot be instanciated, serves as blueprint for derived class template

↓
abstract class driver ——→ master external drivers.
fn(transaction drive)

pure virtual method ⟵ inside abstract class to be overwridden.

Static function → Scope resolution operator.

Static ::

↳ automatic by default

→ deep copy
→ in lab

Memory management in System Verilog - automatically done,

automatic memory management

× multithreading, reentrant environment create problem

Virtual interface

An object cannot be accessed from design & vice versa, virtual interface is used to do so

29/11/2024

Randomization (class) only class can be randomized.

rand <variable_declaration> → scalar, integer, reg, enum, bit, array

→ current_trans.randomize();

rand[c]

↳ cyclic random (constraint cannot be applied to it)

Constraint

→ range, should not-take, arithmatic, < > = ⇔ +, -, /

constraint c1 {noofpacket inside [5...10]}

Implication constrain

→ bidirectional

{rom_address inside [0:2000] → (ram_address inside [2001..4000]}

⇓

if (rom address inside [0:2000])

ram address inside [2001:4000]

Distribution constraint

dist {100 :=1, 200 :=2, 300 :=5}

weighted ratio

100, 200, 300

1 2 5

Generation order

solve (dist) <first> before (dimen) <next> ;

Inline constraint

varname.randomize with(expr)

(eg)

packet.randomize() with { size == (fifo length - fifo_wr_ptr) }

automatically called <
- pre_randomize() method change based on variable
- post_randomize() generate field based on variable.

obj_randomize(); ⇒ return True (or) False

Constraint mode

constrain.constraint_mode(1); // enable

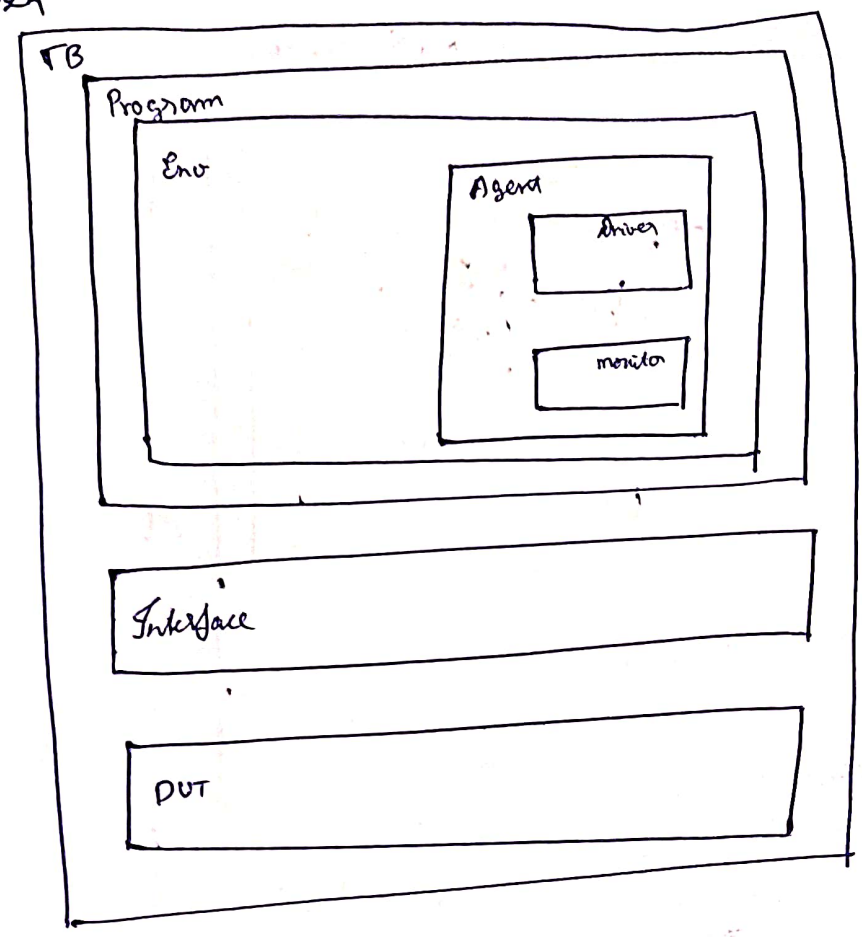
(0); // disable.

Random stability

Random Number Generator → follow thread (or) object for repetability, reproducability.

** Operator overloading removed in 1800-2017

28-11-2024



Constraint
Assertion

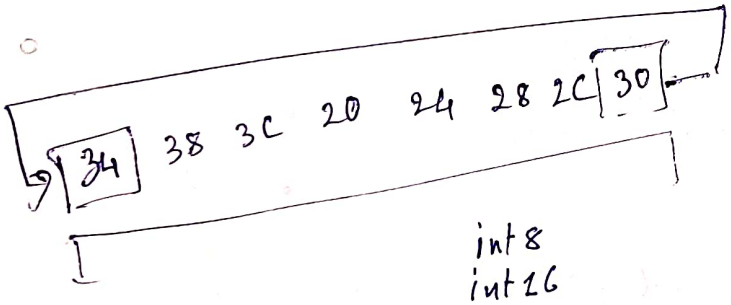
29-11-2024

mailbox, semaphore

transactions, event

← SV-seed

new()
put(msg)
try_put(msg)
peek (ref var)
try_peek (ref var)
get() → gets one removes one



int 8
int 16
int 32
IEEE 754 SP

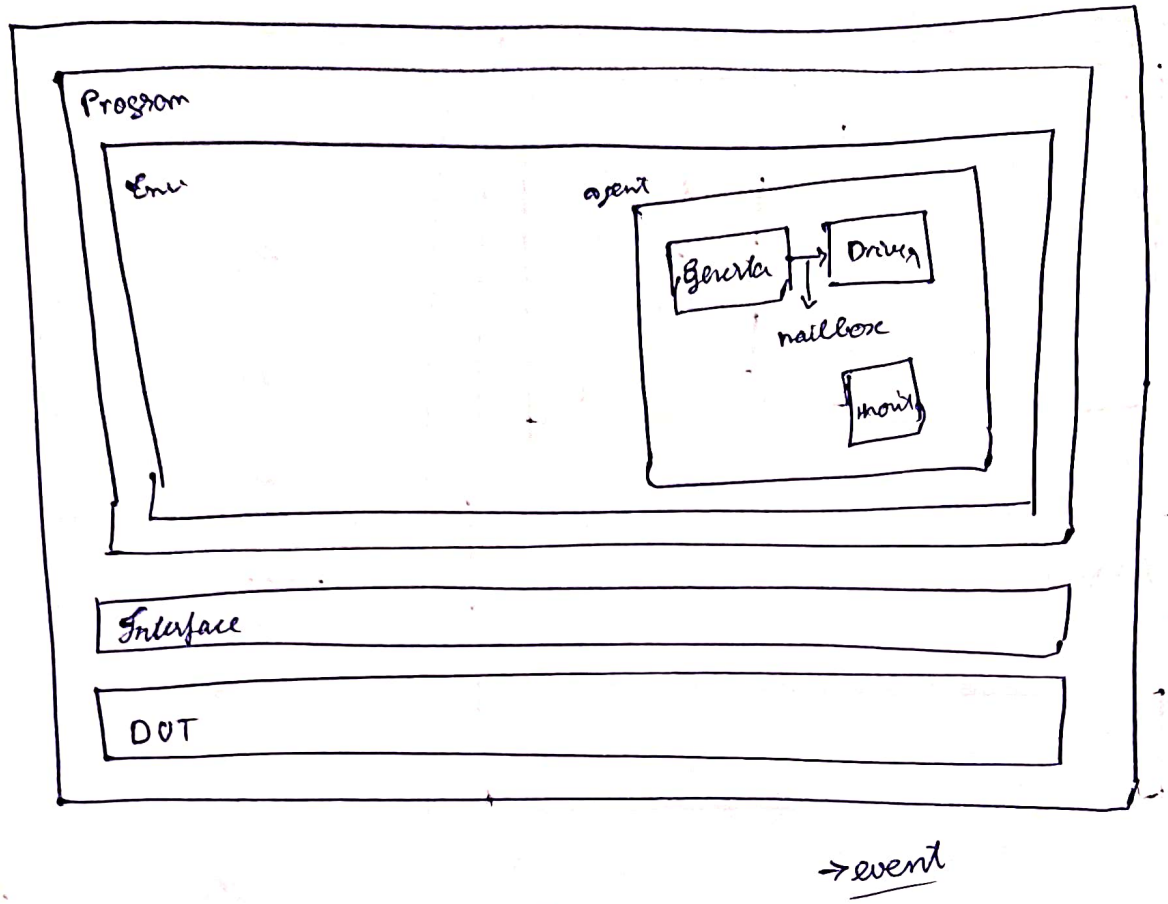
$A_i = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$ $B_i = \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$

$A_i \times B_i = \begin{pmatrix} a_{00}b_{00} + a_{01} * b_{10} & a_{00} \times b_{01} + a_{01} \times b_{11} \\ a_{10}b_{00} + a_{11} \times b_{10} & a_{10}b_{01} + a_{01} \times b_{11} \end{pmatrix}$

M → 8 A → 4

Build
Connect
Run
Cleanup.

30-11-2024



Inside once → component

multiple → transactional class

Types of testbench

→ • reference model (same as DUT)

• Software simulated model (software inferd logic)

• Syntheiable testbench model

02-12-2024

Assertions

Stratified Scheduler.

- discrete event execution model.
- connected threads of execution (or) processes.
- process
  - can be evaluated
  - can be have state
  - responds to i/p & produce o/p
- process are concurrently scheduled.
- change in state of net (or) variable → update event
- process are sensitive to update
- process that are sensitive to that ~~event~~ update is ~~only~~ are considered
- evaluation event → evaluation of process.
- simulation time → ~~fixed~~ time mained by simulator to model actual time
- delta time → single time slot is divided into multiple regions

→ to have clear prediction of event.
→ for particular type executed in order.

Time slot regions.

Prepond, Pre-active, Active, Inactive, Pre-NBA, NBA
Post-NBA, Pre-observed, Observed, Post-observed, Reactive, Re-inactive
Pre-re NBA, ReNBA, Post Re NBA, Re-postponed, Postponed.

- Active - module
- Reactive - testbench
- Observed - assertion

- Sampling of signals before time start.

  Module evaluation & assignment - active region

- Module NBA (non-blocking assignment) - NBA region

- Assertions & checkers - observed region

- Testbench encapsulated in program - reactive region

- Final observation & monitor - postponed region.

Assertions

-> used in timing checks,

Immediate assertions

- executed at the time they are encountered.
- used for simple, one time checks.

```
assert (a>b) else $error("<msg to display.");
```

Concurrent assertions

- monitor over time
- used for checking sequence or temporal properts.

```
property check_abc ;
    @(posedge clk) a |-> ##[1:3] b;
endproperty
assert property (check_abc) else $error("<msg to display");
```

Assertion directive layer

- property asserted (ie) evaluated & checked if true

assert property (check_stable) else $error ("<msg to display");

Property specification layer

property <name>
    <sequence>
endproperty

Sequence layer

- directly specified in a property

sequence <stable>
    a |-> ##[1:3]b ⇒ boolean (or) temporal expression or both.
endsequence

Operator used

-> Delay operator ##<integer>

[*N] -> Repition of event a ##4 c[*3] ⇝ true for 3 consequtive* after 4

[->N] -> goto operator ⟶ non-consequtive

within -> sequence within other.

-> - Immediate implication

|-> -> Overlapping implication (concurrently (or) after antecedent)

|=> -> Non overlapping implication (strictly after antecedent)

Sample value function

$sample - active region value

$rose - posedge

$fell - negedge

$stable - no edge

$changed - change(edge)

$past - returns past value (ie) before N cycles.
(var, N)

Multiclock assertions

@(posedge clk0) sig0 ##1 @(posedge clk1) sig1

Clock resolution

- no explicit clocking event allowed
- multi-clocked sequence & properties are not allowed.
- instance should be singly clocked & clocking identical as clocking block
- infered clk from procedural block supersede default clocking
- explicitly specified clk supersede the default.

03-12-2024

Functional coverage

- intersting scenarios, corner cases & other applicable design cond

Covergroup construct

- set of coverage points
- cross coverage between cover points
- event that synchronize. - Sampling coverage points.
- coverage options
- optional formal argument.

```
covergroup cov1 @m_z; // event sampled at change in m_z
coverpoint m_x;
coverpoint my;
endcovergroup
```

```
coverpoint HREADY {
    bins ready = {1'b1};
    bins not_ready = {1'b0};
}
```

bins → explicitly defined by user or automatically created by System Verilog.

user can define bins for coverpoints with appropriate names & ranges.

↳ bit [3:0] p1;

```
coverpoint p1 {
    bins b1 = {1, [2:5], [6:10]};
    bins b2 = {[1:10], 15};
}
```

Wildcard bins

wildcard bins g12_15[] = {4'b11??};

↑ wildcard entry

↓ each value 1 bin (ie) 4 bins here

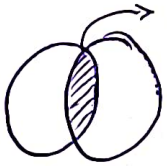
Ignore bins

ignore_bins ignore_1 = {4'b1111};

Illegal bins

illegal_bins illegal_1 = {4'b0000, 4'b1011};

Cross coverage



between two or more coverage points

- cross coverage only include coverpoints
- expressions cannot be used directly
- coverpoints must be explicitly defined

→ bins are created implicitly created as a number of bins as multiple.

```
class cross_cover;
    bit [3:0] a, b, c;
    covergroup cov2 @(posedge clk);
        BC: coverpoint b+c
        axb: cross a, BC;
    endgroup
endclass
```

Transition coverage

trans_list - specify one or more set of ordered integeral value transition of coverage points.

transition bins of real number not allowed

- Single integral value transition

  value 1 => value 2

- Sequence of transitions

  value 1 => value 2 => value 3 => value 4

```
class transition_coverage
    bit [4:1] va;
    covergroup cg @(posedge clk);
      coverpoint va {
        bins sa = (4 => 5 => 6), ([7:9], 10 => 11, 12);
        bins sb[] = (4 ... ... ... );
        bins sc = (12 => 3[=>1]);
                          (->) --> goto operator (ie) 3 times 1 => 1, 1, 1
        bins allother: default sequence; }
    endgroup
endclass
```

Coverage options

- name (string)
- weight (number) // importance
- goal (number) // number of hits
- at_least
- auto_bin_max
- per_instance
- detect_overlap.

```
covergroup g1 (int w) @(posedge clk);
    option.per_instance = 1;
    a : coverpoint a_var { option.auto_bin_max = 128 };
    b : coverpoint b_var { option.weight = w; }
    c1 : cross a_var, b_var;
                a, b
endgroup.
```

Command line arguments

$value$plusargs ("value", value) -> returns 1 if error.

~ vsim work.tb +value=54

04-12-2024

- Randomization using static variable.
- With one master - two slave. { mux to get rdata, decoder for address.

Code coverage

- to identified exercised & not exercised part of code.
- gives quantative measurement of verification process.

Why required?

↳ White box testing
- to find code escaped from testing.
- analyse design aspects covered or not

- levels ⇒ unit, module, subchip level

-

** - coverage goal achived ⇒ end of verification
- not substitute for verification (helps finding defects, DUT functional defects directly / indirectly)

Explained with example code ⇐ 3 different encoding
4'b => 5'b (custom)
↓
Testcase according.

Steps

- Instrumentation
- Analysis
- Report

questa
- vlog -cover <cover_switch> <file_name>
- vsim -coverage <modulename>
- coverage report -file <file_name> -instance <instance_name>

above, if ...

- Statement coverage / Block -> s
- Expression coverage -> e
- Condition coverage - c
- FSM coverage -> f
  - state
  - arc -> transition
  - path / sequence (start -> end)
- Branch coverage -> b
- Toggle coverage - t, x -> extended

- Un-reachable code
  → not exercised on module level [limitation]
- Dead code
  ↳ else not execute.

Code coverage filters

mark lines as ignore

// pragma coverage pragma = off

...

...

// pragma coverage pragma = on

---

1. Verification spaces
2. Data types, operators
3. Procedural Blocks
4. Decision statements, loops
5. Verification, bug, cost of bug, types of bug, finding bug.
6. Task based testbench, component based testbench
7. Object oriented programming. (polymorphism, encapsulation, inheritance)
8. Randomization & constraints
9. Assertions
10. Coverage (Functional coverage & code coverage)