# Excluding bins

- In some cases all the bins may not be of interest, or design should never have a particular bin.

- These are two ways to exclude bins :
  - ignore_bins
  - illegal_bins

# Ignore Bins

- All values or transitions associated with ignore_bins are excluded from coverage.
- Ignored values or transitions are excluded even if they are also included in another bin.

```
bit [3:0] num;
covergroup cg;
coverpoint num  {
bins val [ ]={ [1:15] };                    //7 and 8 are ignored
ignore_bins  ignoreval={ 7, 8 };            //ignore 7 and 8
ignore_bins  ignoretran=(3=>4=>5);   //ignore transition
} endgroup
```

# Ignore Bins

```
bit [2:0] num;

covergroup cg;
coverpoint num  {
option.auto_bin_max=4;
//<0:1> , <2:3>, <4:5>, <6:7>
ignore_bins bins hi={6, 7};
// bins 6 and 7 are ignored from coverage
}
endgroup
```

# Illegal Bins

- All values or transitions associated with illegal_bins are excluded from coverage and run-time error is issued if they occur.

- They will result in a run-time error even if they are also included in another bin.

```
bit [3:0] num;
covergroup cg;
coverpoint num  {
illegal_bins bins illegalval={ 2, 3 };      //illegal bins 2 and 3
illegal_bins bins illegaltran=(4=>5);    //4 to 5 is illegal
//transition
} endgroup
```

# Cross Coverage

- Coverage points measures occurrences of individual values.
- Cross coverage measures occurrences of combination of values.
- Interesting because design complexity is in combination of events and that is what we need to make sure is exercised well.
- Examples:
  - Was write enable 1 when address was 4'b1101.
  - Have we provide all possible combination of inputs to a Full Adder.

# Example1

- Cross coverage is specified between two or more coverpoints in a covergroup.

```
  bit [3:0] a, b;
covergroup cg @ (posedge clk);
cross_cov: cross a , b;
endgroup
```

- 16 bins for each a and b.
- 16 X 16=256 bins for cross_cov

# Example2

- Cross coverage is allowed only between coverage points defined within the same coverage group.

```
bit [3:0] a, b, c;
covergroup cg @ (posedge clk);
cov_add: coverpoint b+c;
cross_cov: cross a , cov_add;
endgroup
```

- 16 Bins for each a, b and c. 32 bins for b + c.
- 16 X 32=512 bins for cross_cov.

# Example3

```
bit [31:0] a;
    bit [3:0] b;
covergroup cg @ (posedge clk);
cova: coverpoint a { bins low [ ]={ [0:9] }; }
cross_cov: cross  b, cova;
endgroup
```

- 16 bins for b. 10 bins for cova.
- 10 X 16=160 bins for cross_cov.

# Cross Coverage

- Cross Manipulating or creating user-defined bins for cross coverage can be achieved using bins select-expressions.

- There are two types of bins select expression :
  - binsof
  - intersect

# binsof and intersect

- The binsof construct yields the bins of expression passed as an arguments. Example:  binsof (X)

- The resulting bins can be further selected by including or excluding only the bins whose associated values intersect a desired set of values.

- Examples:
  - binsof(X) intersect { Y } , denotes the bins of coverage point X whose values intersect the range given by Y.
  - ! binsof(X) intersect { Y } , denotes the bins of coverage point X whose values do not intersect the range given by Y.

# binsof and intersect

- Selected bins can be combined with other selected bins using the logical operators && and ||.

```
bit [7:0] a, b;
covergroup cg @ (posedge clk);
cova : coverpoint a
{
 bins a1 = { [0:63] };
 bins a2 = { [64:127] };
 bins a3 = { [128:191] };
 bins a4 = { [192:255] };
 } endgroup
```

# binsof and intersect

```
covb : coverpoint b
{
 bins b1 = { 0 };
 bins b2 = { [1:84] };
 bins b3 = { [85:169] };
 bins b4 = { [170:255] };
 }
```

# binsof and intersect

```
covc : cross cova, covb
{
 bins c1= !binsof(cova) intersect {  [100:200] };
 //a1*b1, a1*b2, a1*b3, a1*b4
 bins c2= binsof(cova.a2) || binsof(covb.b2);
 //a2*b1, a2*b2, a2*b3, a2*b4
 //a1*b2, a2*b2, a3*b2, a4*b2
 bins c3= binsof(cova.a1) && binsof(covb.b4);
 //a1*b4
 }
 endgroup
```

# Excluding Cross products

- A group of bins can be excluded from coverage by specifying a select expression using ignore_bins.

```
covergroup cg;
cross a, b
{
ignore_bins  ig=binsof(a) intersect { 5, [1:3] };
}
endgroup
```

- All cross products that satisfy the select expression are excluded from coverage even if they are included in other cross-coverage bins of the cross.

# Illegal Cross products

- A group of bins can be marked illegal by specifying a select expression using illegal_bins.

```
covergroup cg (int bad);
cross a, b
{
illegal_bins bins invalid=binsof(a) intersect { bad };
}
endgroup
```

# Coverage Options

- Options can be specified to control the behaviour of the covergroup, coverpoint and cross.
- There are two types of options:
  - Specific to an instance of a covergroup.
  - Specify for the covergroup.
- Options placed in the cover group will apply to all cover points.
- Options can also be put inside a single cover point for finer control.

# option.comment

- Comments can be added to make coverage reports easier to read.

```
covergroup cg;
option.comment=“Cover group for data and address”;
coverpoint data;
coverpoint address;
endgroup
```

# per instance coverage

- If your test bench instantiates a coverage group multiple times, by default System Verilog groups together all the coverage data from all the instances.

- Sometime you would that all coverpoints should be hit on all instances of the covergroup and not cumulatively.

```
covergroup cg;
option.per_instance=1;
coverpoint data;
endgroup
```

# at_least coverage

- By default a coverpoint is marked as hit (100%) if it is hit at least one time.
- Some times you might want to change this to a bigger value.
- Example: If you have a State machine that can handle some kind of errors. Covering an error for more number of times has more probability that you might also test error happening in more than one state.

option.at_least=10

# Coverage goal

- By default a covergroup or a coverpoint is considered fully covered only if it hits 100% of coverpoints or bins.

- This can be changed using option.goal if we want to settle on a lesser goal.

```
bit [2:0] data;
covergroup cg;
coverpoint data;
option.goal=90;     //settle for partial coverage
endgroup
```

# option.weight

- If set at the covergroup level, it specifies the weight of this covergroup instance for computing the overall instance coverage.
- If set at the coverpoint (or cross) level, it specifies the weight of a coverpoint (or cross) for computing the instance coverage of the enclosing covergroup.
- Usage: option.weight=2 (Default value=1)
- Usage: Useful when you want to prioritize certain coverpoints /covergroups as must hit versus less important.

# Example

```
covergroup cg;
a: coverpoint sig_a { bins a0= {0};
                      option.weight=0;  //will not compute to
  //coverage
                      }
b: coverpoint sig_b { bins b1= {1};
                      option.weight=1;
                     }
ab: cross a , b { option.weight=3; }
endgroup
```

# option.auto_bin_max

- Limiting autobins for coverpoints and crosses
- Usage: option.auto_bin_max = <number> (default=64)
- Usage: option.cross_auto_bin_max =<number>
  (default= unbounded)

# Predefined Coverage Methods

| Method (function) | Can be called on | | | Description |
|---|---|---|---|---|
| | covergroup | coverpoint | cross | |
| **void** sample() | Yes | No | No | Triggers sampling of the covergroup |
| **real** get_coverage() | Yes | Yes | Yes | Calculates type coverage number (0...100) |
| **real** get_inst_coverage() | Yes | Yes | Yes | Calculates the coverage number (0...100) |
| **void** set_inst_name(**string**) | Yes | No | No | Sets the instance name to the given string |
| **void** start() | Yes | Yes | Yes | Starts collecting coverage information |
| **void** stop() | Yes | Yes | Yes | Stops collecting coverage information |

# Example1

```
covergroup packet_cg;
coverpoint dest_addr;
coverpoint packet_type;
endgroup

packet_cg pkt;
initial pkt=new;

always @ (pkt_received)
pkt.sample();
```

# Example2

```
covergroup packet_cg;
coverpoint dest_addr;
coverpoint packet_type;
endgroup

packet_cg pkt;
initial pkt=new;

always @ (posedge clk)
if (port_disable) pkt.stop();
else (port_enable) pkt.start();
```

# Coverage system tasks and functions

- $set_coverage_db_name ( name )

  Sets the filename of the coverage database into which coverage information is saved at the end of a simulation run.

- $load_coverage_db ( name )

  Load from the given filename the cumulative coverage information for all coverage group types.

- $get_coverage ( )

  Returns as a real number in the range 0 to 100 that depicts the overall coverage of all coverage group types.

# Cover property

- The property that is used an assertion can be used for coverage using cover property keyword.

```
property ab;
@(posedge clk) a ##3 b;
endproperty

cp_ab: cover property(ab) $info("coverage passed");
```

# Effect of coverage on performance

- Be aware that enabling Functional Coverage slows down the simulation.
- So know what really is important to cover :
  - Do not use auto-bins for large variables.
  - Use cross and intersect to weed out unwanted bins.
  - Disable coverpoint/covergroup during reset.
  - Do not blindly use clock events to sample coverpoint variables, instead use selective sampling() methods.
  - Use start() and stop() methods to decide when to start/stop evaluating coverage.
  - Do not duplicate coverage across covergroups and properties.

- Design clock based  calculator for +,-,*,/ for

  3 bit inputs  dt1 and dt2

  one sel input

  8 bit output
- Create a coverage model for above design
- Create auto bins for sel input
- Create ignore bins for carry
- Create wildcards bins for even and odd for input dt1
- Create a cross covg for sel == * and dt1 and dt2 < 20
- check auto_bin_max for sel
- Use goal method for *  = 50