Functional Coverage
Arturo Salz
Agenda

• Definitions and features
• Coverage definition
  – Coverage group
  – Coverage point
    • Values and Transitions
    • User defined bins
  – Cross coverage
    • Cross product selection and exclusion
• Coverage options
• Procedural control and access to coverage
What is Functional coverage

• Measure of how much of the design specification has been exercised
  – % test plan features
• User-specified
  – Not automatically inferred from the design
• Based on design specification
  – Captures intent
  – Independent of design code or structure
Functional coverage features

- Coverage of variables and expressions
  - Cross coverage
- Automatic and user-defined coverage bins
  - Values, transitions, or cross products
- Filtering conditions at multiple levels
- Flexible coverage sampling
  - Events, Sequences, Procedural
- Directives to control and query coverage
Coverage model : covergroup

New container covergroup : coverage model

• Coverage points
  – variables
  – expressions
  – transitions

• Cross coverage

• Sampling expression : clocking event

• Filtering expressions

• Specify once (like class), use many times
  – Cumulative or per-instance coverage
Declaration of a covergroup

```
covergroup identifier [ ( argument_list ) ] 
[ clocking_event ];
  { coverage_spec_or_option ; } 
endgroup [ : identifier ]
```

```
enum { red, green, blue } color;
bit [3:0] pixel;

covergroup g1 @(posedge clk);
  coverpoint color;
  coverpoint pixel;
  AxC: cross color, pixel;
endgroup
```

- 3 bins for color
- 16 bins for pixel
- 48 cross products
- Sampling event
Coverage sampling event

Sampling can be
- Any event expression - edge, variable
- End-point of a sequence
- Event can be omitted
  - Procedural sampling under user control

```verilog
covergroup cg1 @(posedge clk);
  ...
endgroup

sequence s @(posedge clk) req ##[1:$$] grant; endsequence

covergroup cg21 @(s);
  ...
endgroup
```
Defining Coverage Points

[\text{label :}] \texttt{coverpoint} \text{ expression [ \texttt{iff} (expression) ] }
\{ \text{bins_or_options} \}

Specifies expression (or variable) to sample
- Expression is sampled and accounted in bin(s)
- Number of values/bins can be controlled
  - bins specification
- Optionally filtering expression - \texttt{iff}
- Bins can be grouped using bins specification

\begin{verbatim}
enum \{ \text{red, green, blue} \} color;
covergroup cg @(posedge clk);
  coverpoint color \texttt{iff}(\neg \text{reset});
endgroup
\end{verbatim}
Defining Bins for Coverage Points

• If no state or transition bins explicitly defined, then bins are automatically created
  – Easy-to-use, no effort in defining bins
• Or, user can define state and/or transition bins for each coverage point.
  – Too many values
  – Not all values are interesting or relevant
• Each bin groups a set of values or a set of value transitions associated with a sampled variable or expression
  – Group equivalent values
  – Cover bins, not values
Defining coverage-point bins

- Group specific cover-points under a name
  - Set of values
  - Set of transitions
- default catches undefined values / transitions
- [ ] specifies creation of multiple bins per value
- iff specifies conditional coverage
Defining value coverage bins

```verilog
bit [7:0] v_a, v_b;

covergroup cg @ev1;
  coverpoint v_a + v_b {
    bins a = { [64:127],200 }; // user-defined bins
    bins b[] = { 0,10,100,220 } iff( !reset );
    bins bad = default; // all other values
  }
endgroup
```

- a creates one bin, covered if in the range
- b creates one bin per value: b[0], b[10], b[100], b[220]
  - only covered when reset == 0
- bad catches all other (in one bin)
  - [1:9], [11:63], [128:199], [201:219], [221:255]
Transition coverage bins

trans_range_list -> trans_range_list { -> tras_range_list}

trans_range_list ::= 
    trans_item  
    | trans_item [* repeat_range ]  // consecutive
    | trans_item [*-> repeat_range ]  // goto-repetition
    | trans_item [*= repeat_range ]  // nonconsecutive rept

Subset of property syntax

- {1:8} -> 2 expands to 1->2, 2->2, 3->2, ..., 8->2
- 3->5->{1:2} expands to 3->5->1, 3->5->2
- 2->3[*2:3] expands to 2->3, 2->3->3, 2->3->3->3
- 2->3[*->2] expands to 2->3->...->3->...->3
- 2->4[*=2] expands to 2->4->...->4->...->4 (excluded)
Defining transition bins

```verilog
define bit [7:0] v_a;

covergroup cg @ev1;

coverpoint v_a
{
  bins v = { [10:100], 200 };
  bins t[] = {1:2} -> {2:3}
  bins s = {4 -> 5 -> 6, {[7:8], 9} -> {1, 2}};
  bins bad = default;
}
endgroup
```

- `t` creates one bin per transition: 4 bins
  - 1->2  1->3  2->2  2->3
- `s` creates one bin for all 7 transitions
- `bad` catches all undefined values
  - [0:9], [101:199], [201:255]
The **wildcard** specification treats ?, X, Z as a wildcard for 0 or 1.

```systemverilog
bit [3:0] data;

covergroup cg @(negedge clk);
    coverpoint data {
        wildcard bins p = { 4’b11?? }; // 12, 13, 14, 15
        wildcard bins s[] = (4’b000? -> 4’b001?};
    }
endgroup
```

- **p** creates one bin for the 4 values: 12, 13, 14, 15
- **s** creates one bin for each of the transitions: 0->2, 0->3, 1->2, 1->3
Automatic bin creation

- If omitted, $N$ bins are automatically created
- $N$ is determined:
  - For an enum: $N$ is the cardinality of the enum
  - All others: $N$ is $\min(2^M, \text{auto\_bin\_max})$
    - $M \Rightarrow \# \text{ bits needed to represent the cover-point}$
- If $N < 2^M$
  - Values are uniformly distributed into the $N$ bins
  - Every bin will include $2^M/N$ values
  - Last bin accommodates any slack
- Automatic bins exclude X and Z (2-state only)
- Coverage space is tractable
Excluding values or transitions

- Any set of values or transitions can be explicitly excluded from coverage
  - the `ignore_bins` specification

```verbatim
covergroup g1 @(posedge clk);
  coverpoint a
  {
    ...
    ignore_bins ivals = {7,8};
    ignore_bins itrans = (1->3->5);
  }
endgroup
```

- `ivals` excludes values 7 and 8
- `itrans` excludes the transition 1->3->5
Illegal values or transitions

- Any set of values or transitions can be marked illegal using `illegal_bins`

```verilog
covergroup g1 @(posedge clk);
  coverpoint a {
    ...
    illegal_bins evals = {1,2,3};
    illegal_bins etrans = (4->3->2, 5->2);
  }
endgroup
```

- An Illegal bin hit triggers a run-time error – Even if it is part of another bin
Defining cross coverage

[def: cross coverpoint_list [ iff (expression) ] { select_bins_or_options }]

• Covers two or more coverage points simultaneously
  – Coverage of all combinations of all bins associated with the specified cover-points
  – The Cartesian product of all the sets of coverage-point bins

```systemverilog
enum { red, green, blue } color;
bit [3:0] pixel;

covergroup g1 @(posedge clk);
  coverpoint color;
  coverpoint pixel;
  AxC: cross color, pixel;
endgroup
```

- 3 bins for color
- 16 bins for pixel
- 48 cross products
Defining cross coverage bins

- A cross coverage bin associates a name and a count with a set of cross products
- Cross bins group together sets of cross products

\[
\text{bins\_selection ::= bins name = select\_expression}
\]

\[
\text{select\_expression ::=}
\]

\[
\begin{align*}
\text{select\_condition} & \mid ! \text{select\_condition} \\
\text{select\_expression} \land \text{select\_expression} & \mid \text{select\_expression} \lor \text{select\_expression} \\
( \text{select\_expression} ) & \\
\end{align*}
\]

\[
\text{select\_condition ::=}
\]

\[
\text{binsof ( bins) [ intersect open\_range\_list ]}
\]
Cross coverage bins

```verilog
bit [7:0] v_a, v_b;

covergroup cg @clk;
    a: coverpoint v_a {
        bins a1 = { [0:63] };
        bins a2 = { [64:127] };
        bins a3 = { [128:191] };
        bins a4 = { [192:255] };
    }
    b: coverpoint v_b {
        bins b1 = {0};
        bins b2 = { [1:84] };
        bins b3 = { [85:169] };
        bins b4 = { [170:255] };
    }
    c : cross v_a, v_b ;
endgroup
```

16 cross products:
<a1,b1>...<a1,b4>
<a4,b1>...<a4,b4>
Cross coverage bins

```verilog
bit [7:0] v_a, v_b;

covergroup cg @clk;
    a: coverpoint v_a {
        bins a1 = { [0:63] };
        bins a2 = { [64:127] };
        bins a3 = { [128:191] };
        bins a4 = { [192:255] };
    }
    b: coverpoint v_b {
        bins b1 = {0};
        bins b2 = { [1:84] };
        bins b3 = { [85:169] };
        bins b4 = { [170:255] };
    }
    c : cross v_a, v_b {
        bins c1 = ! binsof(a) intersect { [100:200] };
        bins c2 = binsof(a.a2) || binsof(b.b2);
        bins c3 = binsof(a.a1) && binsof(b.b4);
    }
endgroup
```

4 cross products:
- `<a1,b1>,<a1,b2>`
- `<a1,b2>,<a1,b4>`

7 cross products:
- `<a2,b1>...<a2,b4>`
- `<a1,b2>...<a4,b2>`

1 cross product:
- `<a1,b4>`
Exclusion cross products

- Select expressions can be used to exclude or specify cross products as illegal

```verilog
covergroup yy;
cross a, b
{
    ignore_bins x = binsof(a) intersect {5,[1:3]};
    illegal_bins x = binsof(a) intersect {[25:$]};
}
endgroup
```

- Illegal bins take precedence over all others
- Excluded bins are never included
Generic Coverage groups

• Generic coverage groups can be written by passing their traits as arguments to the constructor.

```verilog
covergroup rg (ref int ra, int low, int high ) @(clk);
    coverpoint ra // sample variable passed by reference
    {
        bins good = { [low : high] };
        bins bad[] = default;
    }
endgroup
```

• good creates one bin, for the range [low : high]
• bad creates one bin per value outside that range

```verilog
int A, B;
rg c1 = new( A, 0, 50 ); // cover A in range 0 to 50
rg c2 = new( B, 120, 600 ); // cover B in range 120 to 600
```
Coverage Group in classes

- Coverage groups may be embedded in class
  - Integrated with object oriented paradigm
  - Intuitive and simple to cover data members
    - Including private data members
  - Other class members can be seamlessly used in coverage specification
class xyz;
    bit [3:0] m_x;
    int m_y;
    bit m_z;

    covergroup cov1 @m_z; // embedded covergroup
        coverpoint m_x;
        coverpoint m_y;
    endgroup

    function new(); cov1 = new; endfunction
endclass
coveragegroup g1 (int w, string iComment) @(posedge clk) ;
  // track coverage information for each instance of g1
  option.per_instance = 1;
  option.comment = iComment; // comment for each instance of g1

  a : coverpoint a_var
  {
    option.auto_bin_max = 128;
  }
  b : coverpoint b_var;
  {
    // contributes w times more than a and c1
    option.weight = w;
  }
  cl : cross a_var, b_var ;
endgroup

- creates 128 bins max
- contributes w times more
Options for control

• weight
  – for computing weighted mean
• goal
  – target goal for group/point/cross
• name
  – name for the covergroup instance
• at_least
  – minimum number of hits for a bin
• per_instance
  – keep per instance data in addition to the cumulative coverage
Coverage Control

- Covergroup and covergroup instance methods allow control and access to the coverage data
- `void sample()`
  - Procedurally control sampling
- `real get_coverage()`
  - Obtains cumulative coverage
- `real get_inst_coverage()`
  - Obtains instance coverage
Procedural sampling

define

g1 = new;

Sample for coverage at this point
Coverage Control

• Methods to start and stop collection
  – start()
  – stop()

• System function to retrieve overall coverage
  – $get_coverage()

• System tasks to name, load and save coverage database
  – $set_coverage_db_name()
  – $load_coverage_db()
Thank you