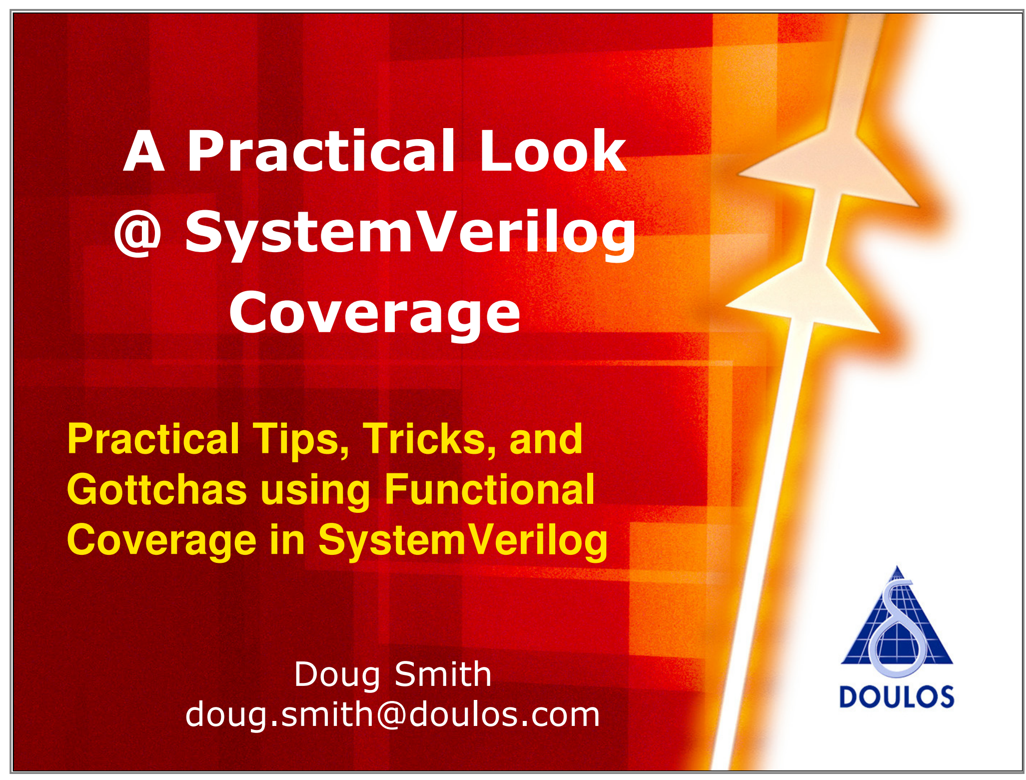
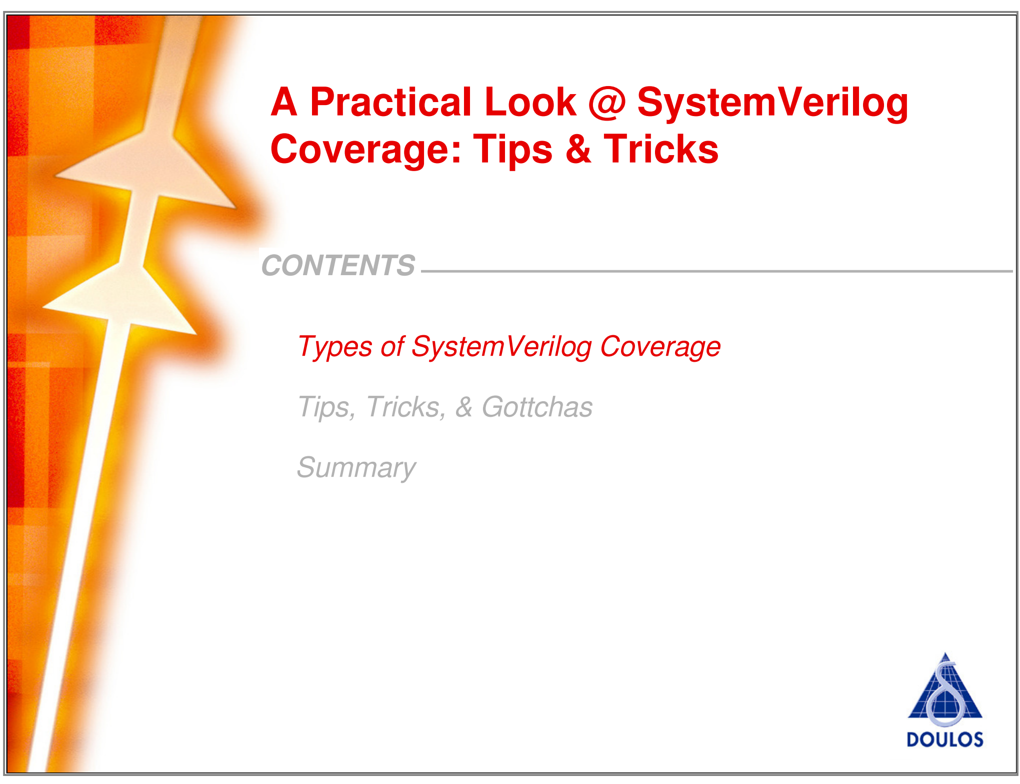


Notes

Often I am asked by students how to accomplish various tasks using SystemVerilog coverage. For example, students often ask, "How can I use my coverage to feedback into my random constraints?" So the purpose of this presentation is to provide a few practical tips and tricks using SystemVerilog coverage as well as a few gottchas to avoid.

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Let's first take a quick look at the 2 types of functional coverage provided by SystemVerilog.

Copyright © 2009 by DOULOS. All Rights Reserved

Cover Property vs Covergroup

- Cover properties …
  - Use SVA temporal syntax
  - Can use the same properties that assertions use
  - Not accessible by SV code
  - Placeable in structural code only
- Covergroups …
  - Record values of coverpoints
  - Provide functional crosses of coverpoints
  - Accessible by SV code and testcases
  - Placeable in both objects and structural code

Copyright © 2009 by Doulos. All Rights Reserved

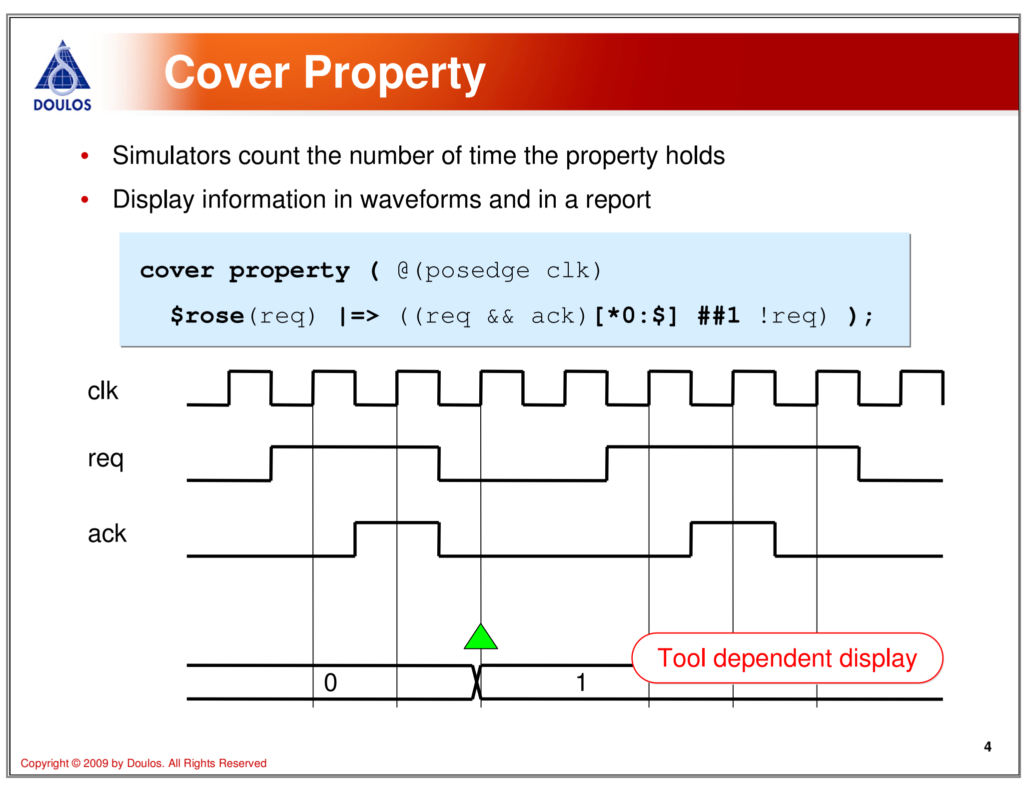
3

Notes

Functional coverage comes in 2 flavors in SystemVerilog. The first type, cover properties, uses the same temporal syntax used by SystemVerilog assertions (SVA). This temporal syntax is used by properties and sequences, which can either be used by the SystemVerilog assert, assume, or cover statements. The advantage of this is that for the same amount of effort you get double the mileage--i.e., the same properties can be used for both assertions and collecting functional coverage. Unfortunately, cover properties can only be placed in structural code (i.e., modules, programs, or interfaces) and can not be used in class-based objects.

The second type of functional coverage is a covergroup. Covergroups record the number of occurrences of various values specified as coverpoints. These coverpoints can be hierarchically referenced by your testcase or testbench so that you can query whether certain values or scenarios have occurred. They also provide a means for creating cross coverage (more on that in a subsequent slide). Unlike cover properties, covergroups may be used in both class-based objects or structural code.

Copyright © 2009 by DOULOS. All Rights Reserved



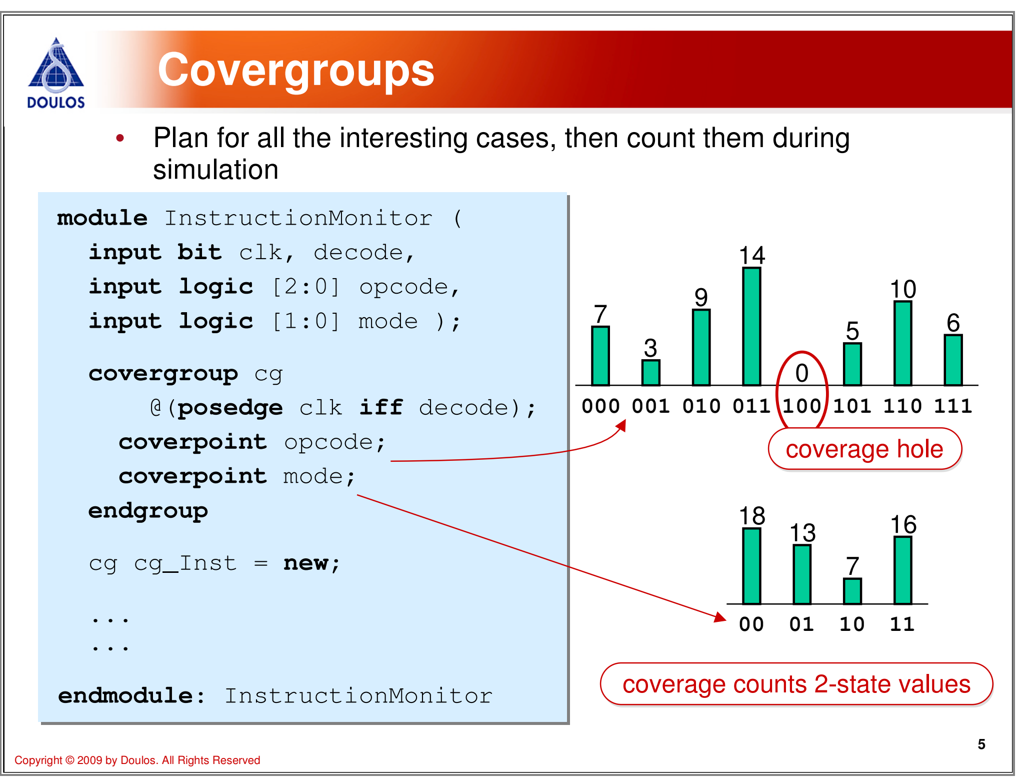
Notes

Here is an example of a cover property. Notice, the cover property uses the same temporal syntax as SVA. This example can be read as:

"When req rises, that implies 1 cycle later that both req and ack are high for 0 or more cycles followed by a cycle of req low"

The simulator keeps track of how many times this sequence occurs and you can view it in your simulator waveform or coverage report.

Copyright © 2009 by DOULOS. All Rights Reserved



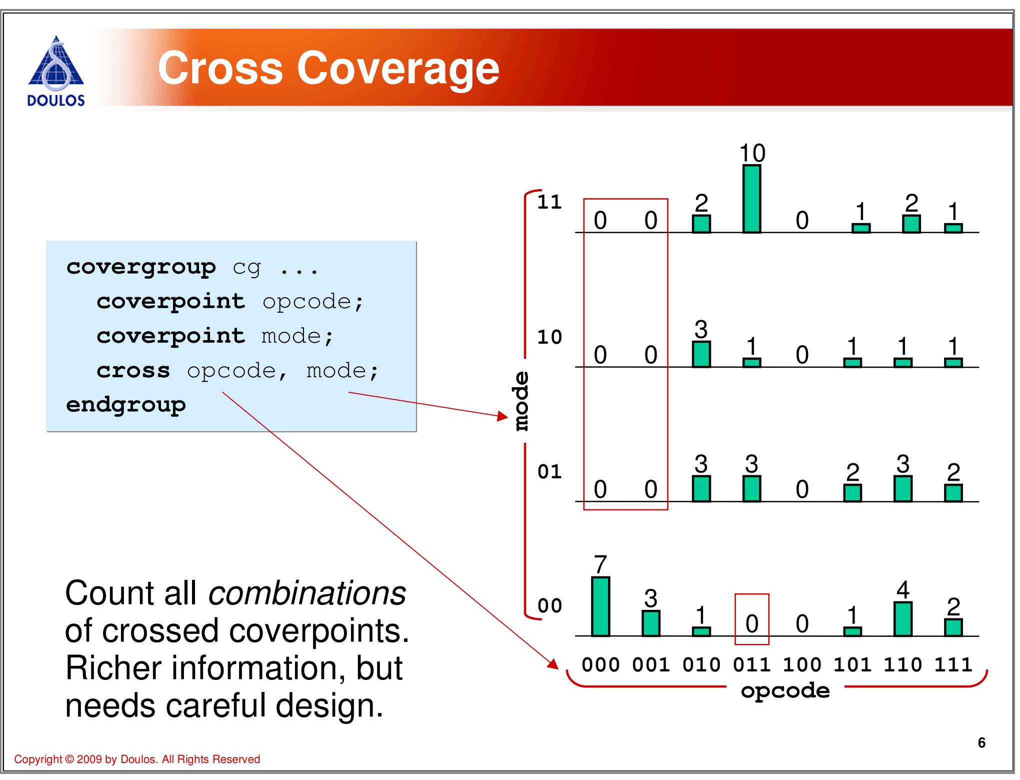
Notes

Here is an example of a covergroup. When defining a covergroup, you need to give it a name ("cg" in this example) and optionally provide a sampling event, which in this case is the positive edge of "clk" qualified by the decode signal. In other words, then a valid instruction occurs ("decode" asserted), then sample the values on the opcode and mode signals.

Since opcode has $2^3 = 8$ possible values, 8 bins or buckets will be created to keep track of the number of times each value occurs. For the mode input, there are $2^2=4$ possible values so 4 bins will be created.

Defining the covergroup alone will not start the coverage collection. Rather, a covergroup needs to be instantiated using the "new" operator and given an instance name. Inside a class, an instance name is not required and the new operator is called on the covergroup instead of the class constructor.

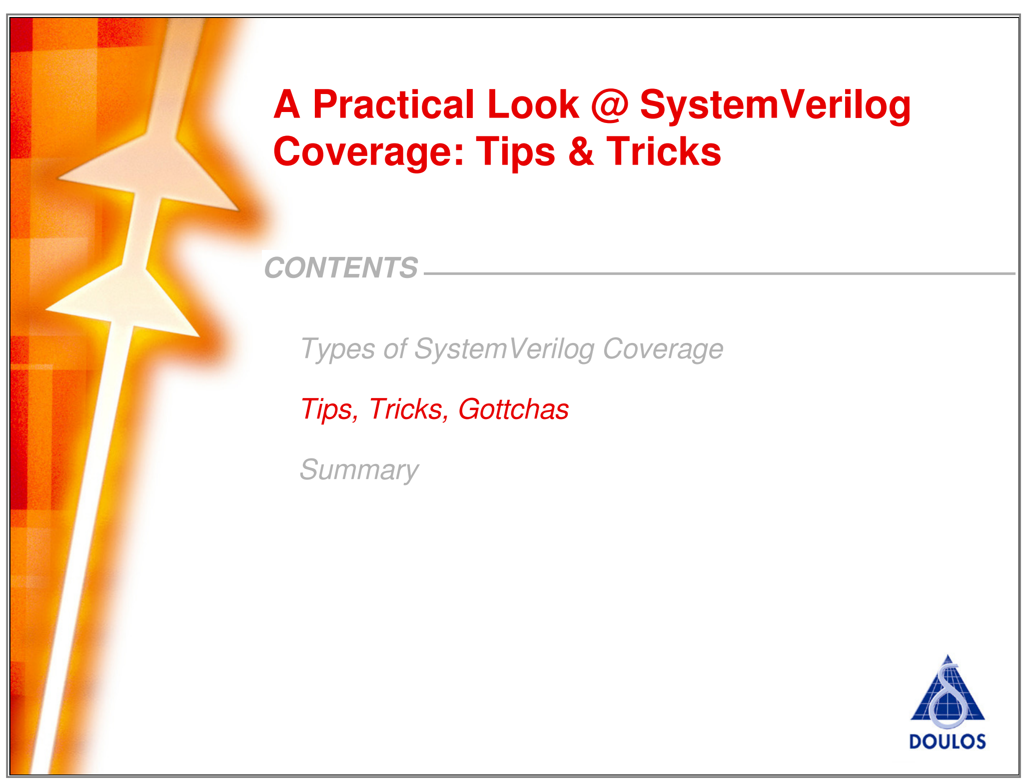
Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Where coverage starts to become really interesting is when we cross different coverpoints to see when different values occur at the same time. In this example, we are crossing all occurrences of the different opcodes occurring at the same time as the four possible mode values. The zeros in the matrix reveal coverage holes--values that have either not been testing, generated, or possible values that are invalid or undefined.

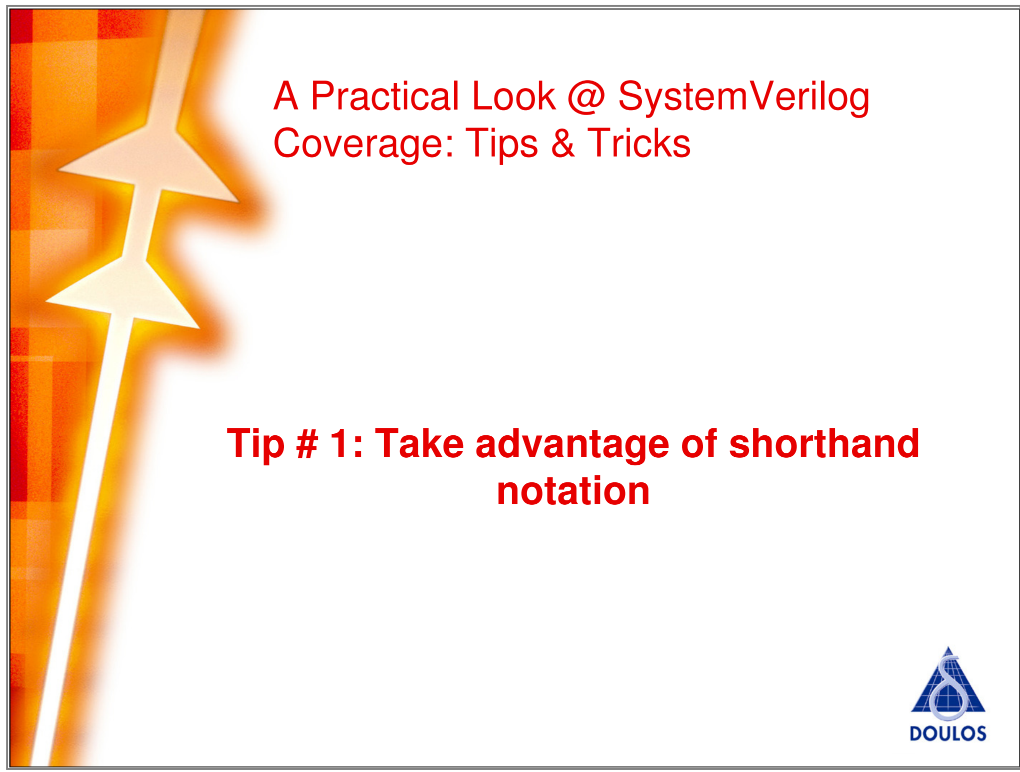
Copyright © 2009 by DOULOS. All Rights Reserved



Notes

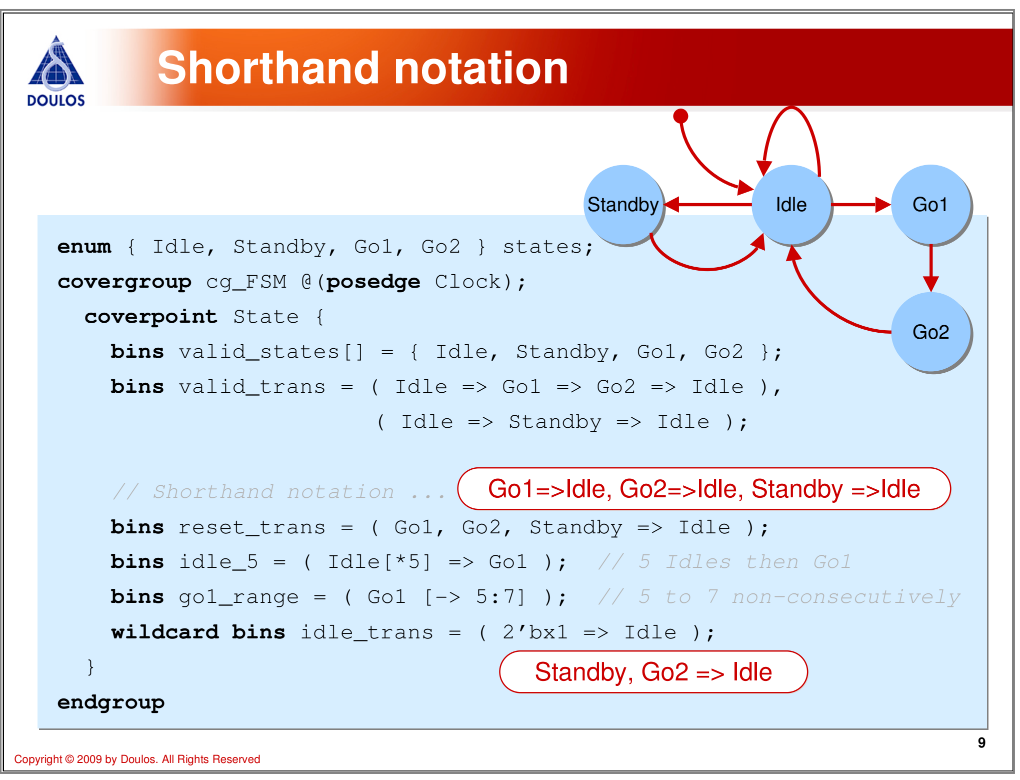
That is a quick overview of SystemVerilog coverage. Now let's have a look at some tips, tricks, and gottchas to avoid when using SystemVerilog functional coverage.

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

SystemVerilog defines many concise ways to define the coverage that you are looking for. Here's an example of a state machine and we are going to define transitional coverage--i.e., a record of the transitions from one state to the next. Notice, to define transitional coverage, use the (=>) syntax.

Some shorthand notations include:

(1) S1, S2, S3 => N1, N2

which translates into

S1=>N1, S1=>N2, S2=>N1, S2=>N2, S3=>N1, S3=>N2

(2) [*N] - repetition operator

(3) [->N:M] - non-consecutive operator (i.e., so many occurrences over an indeterminate period of time)

(4) wildcards … ? - multiple matches

Copyright © 2009 by DOULOS. All Rights Reserved

Notes

Copyright © 2009 by DOULOS. All Rights Reserved

Arguments

```
covergroup cg (ref int v, input string comment);
  coverpoint v;

  option.per_instance = 1;
  option.weight = 5;
  option.goal = 90;
  option.comment = comment;
endgroup

int a, b;

cg cg_inst1 = new(a, "This is cg_inst1 - variable a");
cg cg_inst2 = new(b, "This is cg_inst2 - variable b");
```

Same definition - multiple uses

Copyright © 2009 by Doulos. All Rights Reserved

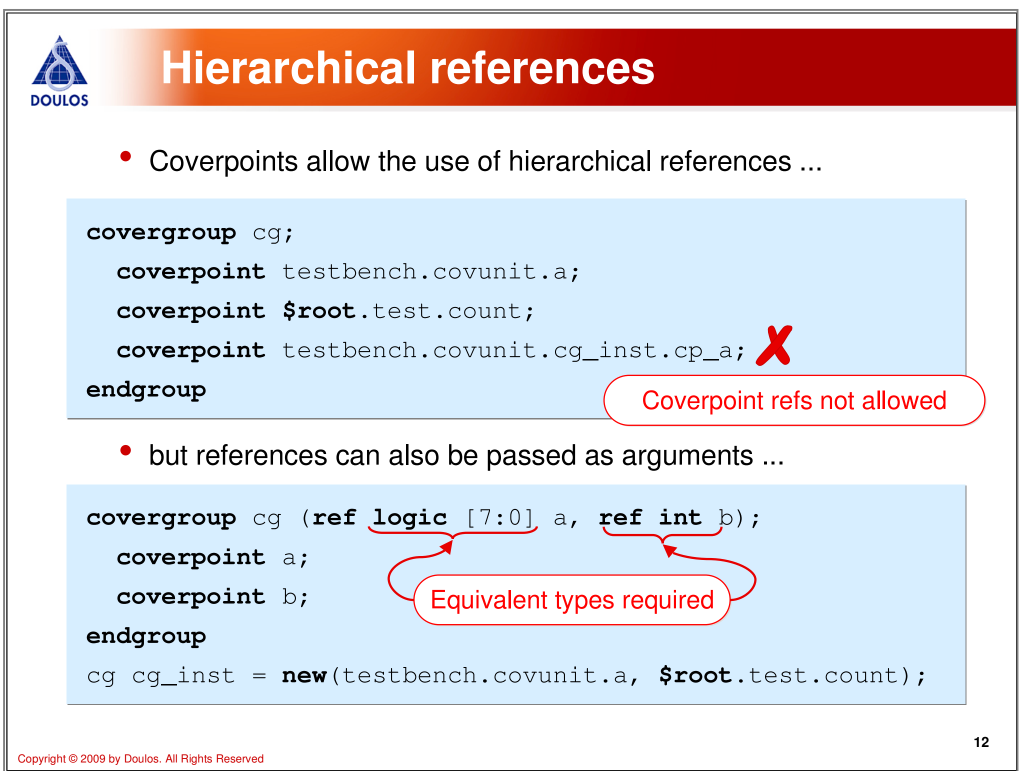
11

Notes

Covergroups can also include arguments (using the same syntax as functions or tasks). In this example, we have added an argument for "v" so that we can pass into the covergroup whatever signal or variable that we want to cover. Notice, we pass the argument by reference by using the "ref" keyword. Likewise, we can pass other arguments like strings that we can use in the covergroup options.

Once we have added arguments, now we can create multiple instances and pass into them the signal/variable we want to cover by passing them in the call to "new". This allows us to reuse your covergroup definitions.

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Covergroups can contain coverpoints to hierarchical references, which can be quite useful. However, they cannot be references to other coverpoints as the top example illustrates. Unfortunately, when we start using hardcoded hierarchical references, our covergroup (and consequently, our testbench) is not as flexible or reusable as it could be. Instead, we could define arguments to our covergroup and then pass hierarchical references into the covergroup when it is instantiated. The instantiation could be done in a testcase or elsewhere so that now the covergroup is much more flexible.

Copyright © 2009 by DOULOS. All Rights Reserved

Notes

Copyright © 2009 by DOULOS. All Rights Reserved

Type options

- Type options apply to the entire covergroup type

```
covergroup cg @(posedge clk);
  type_option.weight = 5;   // Weight in calculation
  type_option.goal = 90;    // Percentage of coverage
  type_option.strobe = 1;   // Sample in postponed region
  cp_a: coverpoint a {
       type_option.comment = comment;
  };
  coverpoint b;
endgroup
```

```
cg::type_option.goal = 100;
cg::cp_a::type_option.weight = 80;
```

Requires constant expressions

Copyright © 2009 by Doulos. All Rights Reserved

14

Notes

Covergroups have many options that can be specified. Type options apply to the entire covergroup type and can only be set when the covergroup is declared or by using the scope resolution operator (::). Type options are specified using the type_option covergroup member. There are 4 type options--weight, goal, strobe, and comment--where

weight = weight of coverage in the coverage calculation

goal = percentage of coverage to reach (this determines whether you see a red, amber, or green color in your coverage report)

strobe = sample the coverage values once everything is stable right before moving on to the next simulation time step (i.e., the postponed simulator region)

comment = string comment

Copyright © 2009 by DOULOS. All Rights Reserved

Per instance options

```
covergroup cg @(posedge clk);
  option.per_instance = 1;     // Turns on these options
  option.weight = 5;    // Weight in coverage calculation
  option.goal = 90;     // Percentage of at_least value
  option.at_least = 10; // Number of occurrences to reach
  option.comment = comment;

  a: coverpoint a { option.auto_bin_max = 128; };
  b: coverpoint b { option.weight = 50; };

endgroup
```

These options require per_instance = 1

```
cg g1 = new;
g1.option.goal = 100;
g1.a.option.weight = 80;
```

Instance can be used

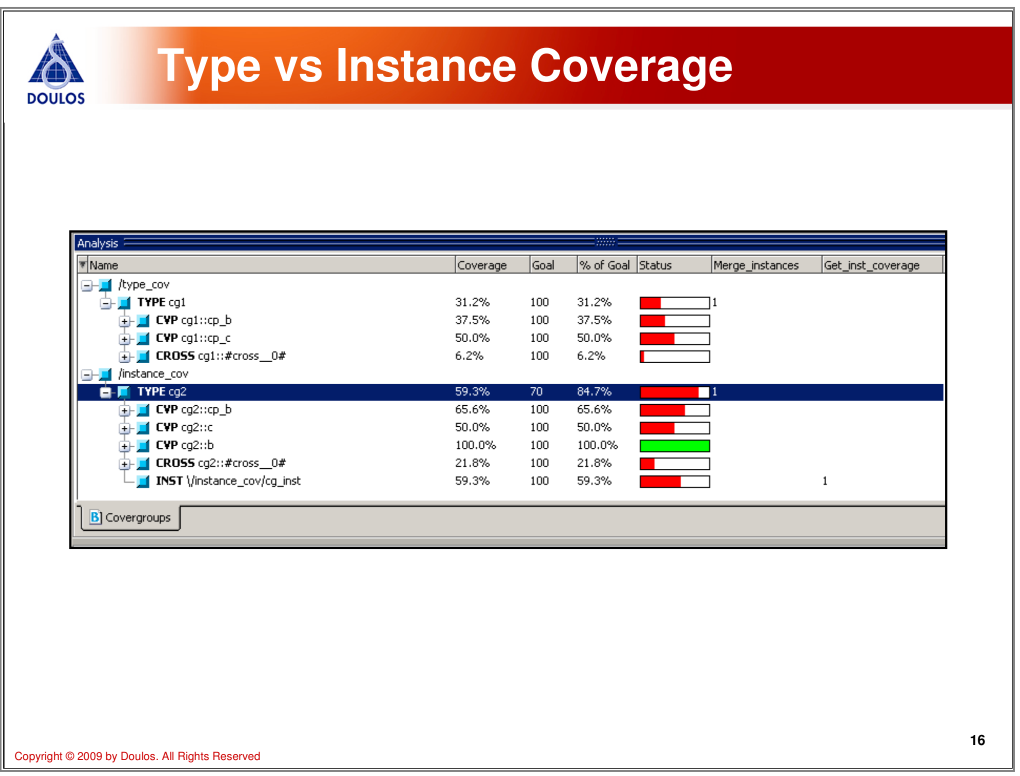
Copyright © 2009 by Doulos. All Rights Reserved

15

Notes

In general, coverage is cumulative unless you specify the per_instance option. When this option is set, then coverage is collected separately for each instance of the covergroup. There are many per_instance options as shown above. One worth pointing out is the at_least option. This option specifies the number of occurrences to see covered and is used to determine if the covergroup goal has been met. Notice, per_instance options are specified using the "option" covergroup member and most can be used with covergroups, coverpoints, and crosses.

Copyright © 2009 by DOULOS. All Rights Reserved



Name	Coverage	Goal	% of Goal	Status	Merge_instances	Get_inst_coverage
/type_cov						
TYPE cg1	31.2%	100	31.2%		1	
CVP cg1::cp_b	37.5%	100	37.5%			
CVP cg1::cp_c	50.0%	100	50.0%			
CROSS cg1::#cross__0#	6.2%	100	6.2%			
/instance_cov						
TYPE cg2	59.3%	70	84.7%		1	
CVP cg2::cp_b	65.6%	100	65.6%			
CVP cg2::c	50.0%	100	50.0%			
CVP cg2::b	100.0%	100	100.0%			
CROSS cg2::#cross__0#	21.8%	100	21.8%			
INST \/instance_cov/cg_inst	59.3%	100	59.3%			1

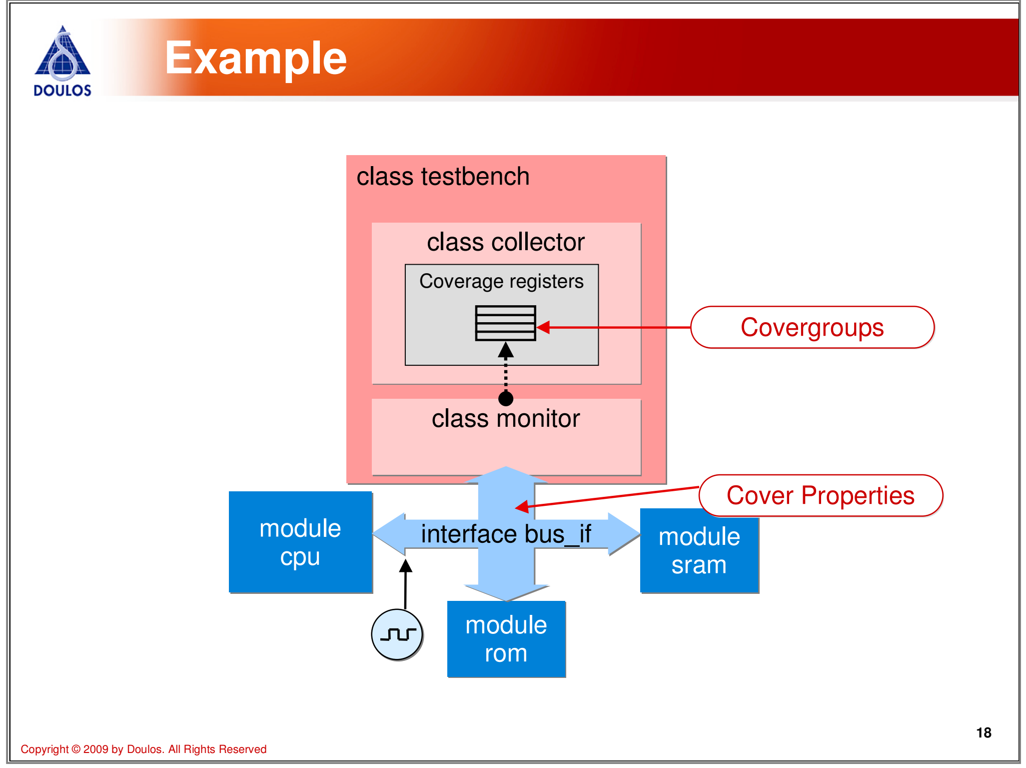
Notes

Here is an example of the difference between type and per_instance coverage as shown in a simulator.

Copyright © 2009 by DOULOS. All Rights Reserved

Notes

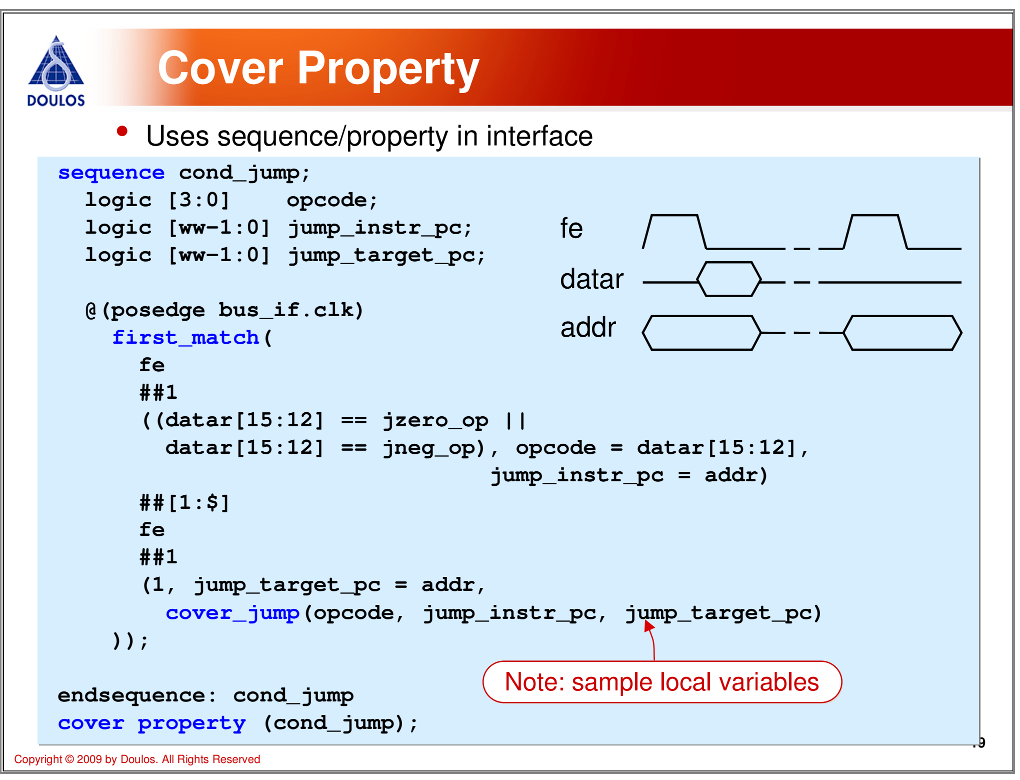
Copyright © 2009 by DOULOS. All Rights Reserved



Notes

In this simple example, the CPU, ROM, and SRAM are connected using a structural SystemVerilog interface called bus_if. Because it is structural, we can place cover properties in the interface and use it to detect when a sequence occurs. When the sequence happens, the bus transaction can be thrown over to the class-based environment by the cover property where it can be recorded in a covergroup structure. Once recorded by the covergroup, interesting things like cross coverage can be done with it as well as gathering coverage feedback for test stimulus. This example will show how to combine the two together.

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

This slide illustrates a sequence used to detect a jump instruction, grab the instruction address, and grab the following target jump address. Using SystemVerilog's temporal syntax, this sequence can be using written instead of writing a more complicated state machine in the class-based monitor. Local variables are used in the sequence to store the opcode and addresses as they occur and then send them to the class-based monitor using the function cover_jump(). The cover property syntax is used to create a process that waits for the sequence to occur.

Copyright © 2009 by DOULOS. All Rights Reserved

Transfer Coverage to Monitor

```
typedef struct packed  {
  logic [3:0]     opcode;
  logic [ww-1:0] jump_instr_pc,
  logic [ww-1:0] jump_target_pc;
} jump_data_t;
```

```
interface basic_bus_if;
 jump_data_t jump_data;
 bit   jump_trig = 0;

 function void cover_jump( logic [3:0] op,
             logic [ww-1:0] jump_instr_pc, jump_target_pc );
   jump_data = {op, jump_instr_pc, jump_target_pc};
   jump_trig = ~jump_trig;
 endfunction
 ...
```

Interface function

```
class monitor extends ...;
  task process_jump();
    forever begin
      jump_data_t t; jump_xact tr;
      @(m_bus_if.mon_xact.jump_trig)
      t = m_bus_if.mon_xact.jump_data;
      tr = new(t.opcode, t.jump_instr_pc, t.jump_target_pc);
      cov_collector.write(tr);     // Send to covergroup
    end
  endtask: process_jump
  ...
```

Bus monitor task

Copyright © 2009 by Doulos. All Rights Reserved

20

Notes

The cover_jump() function also lives in the interface along with the sequence. It is a simple function that simply takes in the opcode, instruction address, and target address and packs them into a structure. A trigger flag called jump_trig is used to signify to the monitor that the sequence has occurred. A flag is used instead of an event to avoid simulator issues with events through virtual interfaces.

The class-based monitor waits on the jump_trig toggling to occur. Then it grabs the packed structure, loads it into a transaction, and then writes it over to the coverage collector using its write() function.

Copyright © 2009 by DOULOS. All Rights Reserved

Record Coverage with Covergroup

```
class collector extends ...;
  ...
  virtual function void write( input jump_xact t );
    m_cov.opcode = t.opcode;
    m_cov.jump_instr_pc  = t.jump_instr_pc;
    m_cov.jump_target_pc = t.jump_target_pc;
    m_cov.jump_delta = m_cov.jump_target_pc - m_cov.jump_instr_pc - 1;
    m_cov.cov_jump.sample();
  endfunction
  ...
```

```
covergroup cov_jump;
   coverpoint jump_delta {
      bins no_jump         = { 0 };
      bins short_jump_fwd  = { [1:15] };
      bins long_jump_fwd   = { [16:2**ww-1] };
      bins short_jump_back = { [-15:-1] };
      bins long_jump_back  = { [-2**ww+1:-16] };
      option.at_least = 4;
   }
endgroup: cov_jump
```

Copyright © 2009 by Doulos. All Rights Reserved

21

Notes

In the coverage collector class, the write function receives the transaction from the monitor. It then calculates the jump address distance and invokes the covergroup's built-in sample() method. The covergroup snapshots the jump_delta and places the values into the corresponding bins.

Copyright © 2009 by DOULOS. All Rights Reserved

Advantages

- Cover property ...
  - Protocol defined in the interface (everything kept together)
  - Protocol defined using temporal syntax--not a custom FSM
- Covergroup …
  - Provides additional coverage options
  - Provides cross coverage
  - Accessible by testbench or testcase (coverage feedback)

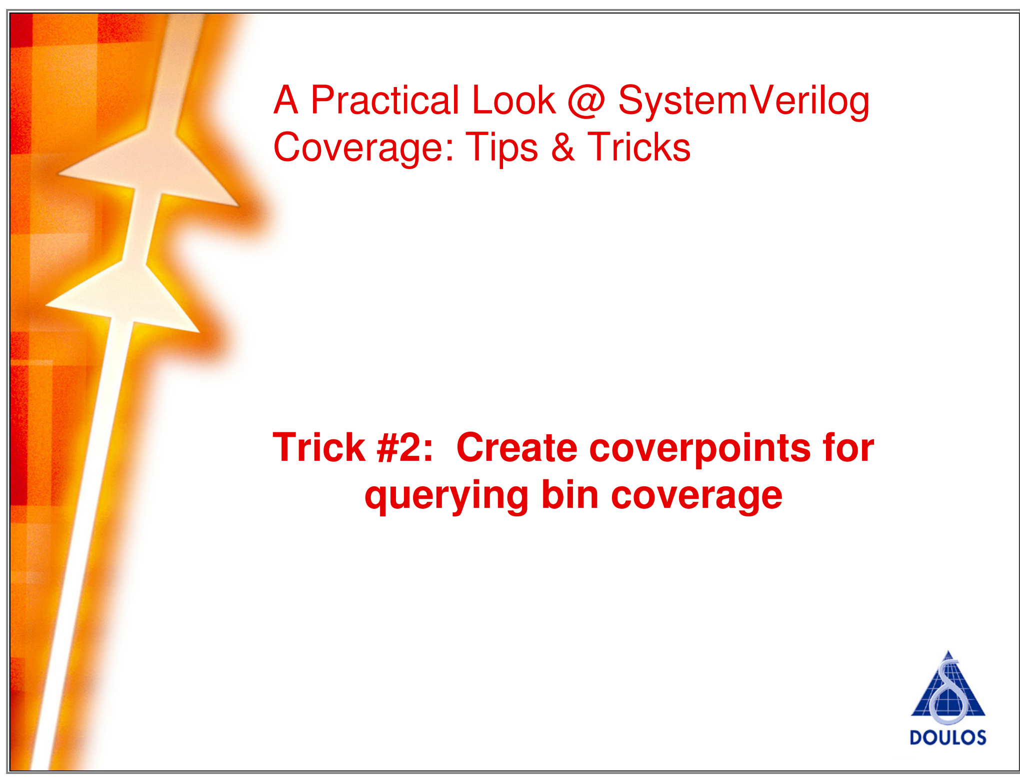
Copyright © 2009 by Doulos. All Rights Reserved

22

Notes

Using this approach allows you to have the best of both worlds. The temporal syntax can be used to create the FSM to monitor the bus protocol, and then covergroups can be used in the class-based environment to record the information. Once the information is in the covergroup, then cross coverage can be created or coverage used for feedback into test cases or the testbench.

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Copyright © 2009 by DOULOS. All Rights Reserved

Notes

Built-in to all covergroups, coverpoints, and crosses is a function called get_coverage(). Get_coverage() returns a real number of the percentage of coverage that has been covered.

In the top example, the sample() method is being used to manually sample the coverage values. The coverage percentage is then used to determine if the goal of 90.0% has been met and if so then stop collecting coverage.

In the bottom example, the current percentage of coverage is being used to determine the weighting in the randcase statement. The function $rtoi() turns the coverage percentage into an integer value so an integer expression can be calculated for the randcase weightings.

Copyright © 2009 by DOULOS. All Rights Reserved

Querying bin coverage

```
covergroup cg;
  coverpoint i {
    bins zero     = { 0 };
    bins tiny     = { [1:100] };
    bins hunds[3] = { 200,300,400,500,600,700,800,900 };
  }
endgroup
```

- **get_coverage()** does not work on bins

```
cov = cg_inst.i.zero.get_coverage();
```

✗ Not allowed

Copyright © 2009 by Doulos. All Rights Reserved

25

Notes

SystemVerilog does not allow querying coverage on individual coverage bins. This is unfortunate, especially if coverage feedback is needed to know which transaction values have occurred since that level of detail would be specified using specific coverage bins.

Copyright © 2009 by DOULOS. All Rights Reserved

Create coverpoints for each bin

```
covergroup instr_cg;
  op_nop :
  coverpoint instr_word[15:12] { bins op = { nop_op }; }

  op_load :
  coverpoint instr_word[15:12] { bins op = { load_op };}

  op_store :
  coverpoint instr_word[15:12] { bins op = { str_op }; }

  op_move :
  coverpoint instr_word[15:12] { bins op = { move_op }; }
  ...
endgroup
```

- Now **get_coverage()** can be used ...

```
cov = cg_inst.op_nop.get_coverage();
```

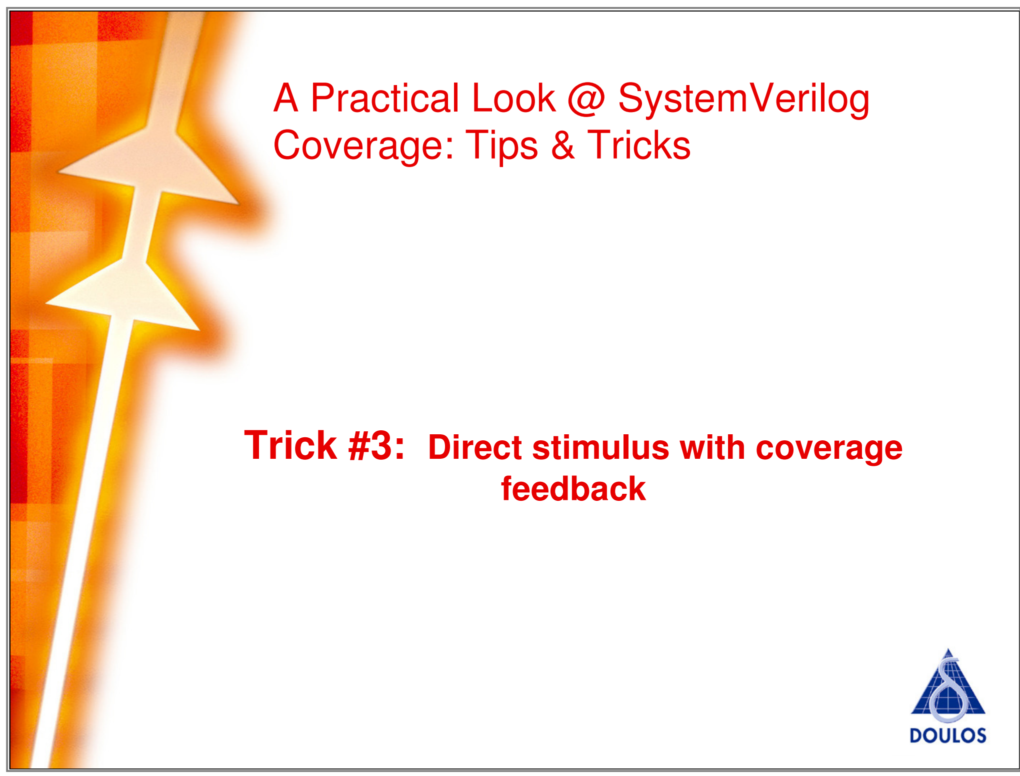
Copyright © 2009 by Doulos. All Rights Reserved

26

Notes

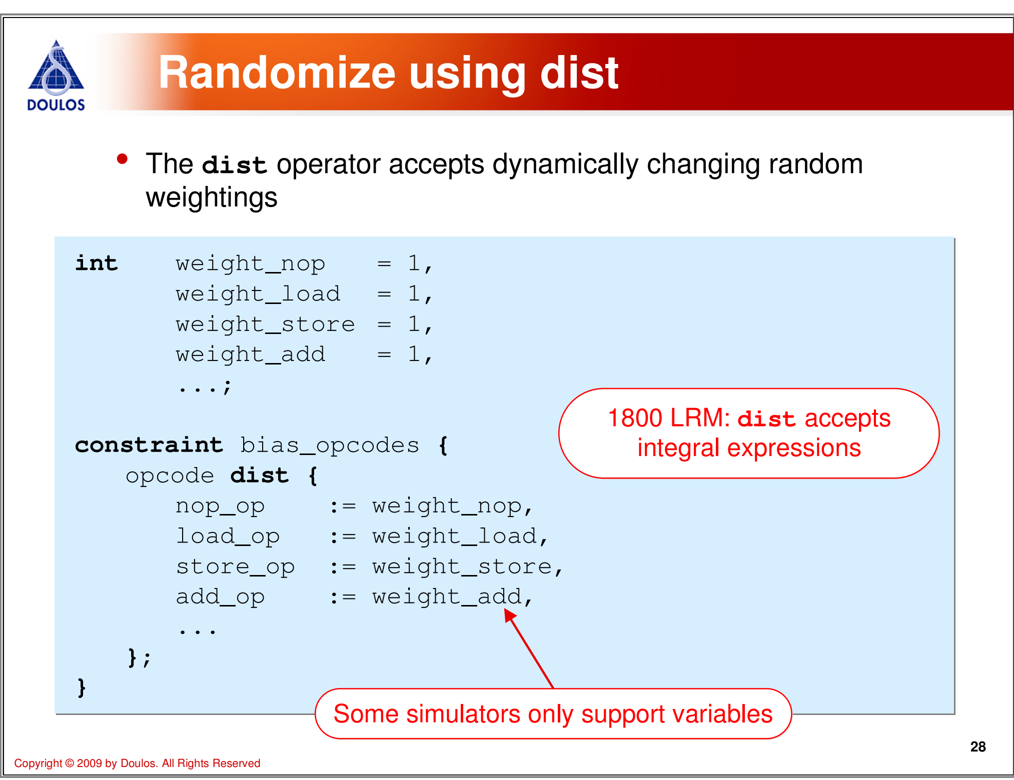
Fortunately, there is a workaround to not being able to query coverage on an individual bin. Instead, each value of interest can be turned into a unique coverpoint so that the get_coverage() function can be called on each value of interest. This syntax is somewhat cumbersome and tedious, but it accomplishes this goal, which is particularly needed to feedback information into random constraints (see the next trick).

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Often times, engineers want to feedback coverage information into their constrained random stimulus generation. Fortunately, SystemVerilog provides a constraint option that accepts a distribution weighting called "dist". With the dist constraint, you can specify the probably that a particular value will occur.

This weighting can be an expression except not all simulators support expressions. So variables can be created for each value's weighting. In the example above, all values have an equal weighting of 1 at simulation startup.

As simulation progresses, these value weightings will be updated to affect the randomization of the opcode stimulus.

Copyright © 2009 by DOULOS. All Rights Reserved

Use pre_randomize to set weights

- `pre_randomize()` sets the weighting used by the `dist`

```
function int calc_weight( opcode_t op );
   real cov;
   case ( op )   // Grab coverage (see Trick #2)
      nop_op:   cov = covunit.cg.op_nop.get_coverage;
      load_op:  cov = covunit.cg.op_load.get_coverage;
      store_op: cov = covunit.cg.op_store.get_coverage;
      ...
   endcase
   calc_weight = 100 - $rtoi(cov) + 1;
endfunction : calc_weight

function void pre_randomize();// Set dist weighting
   weight_nop   = calc_weight(nop_op);
   weight_load  = calc_weight(load_op);
   weight_store = calc_weight(store_op);
   weight_add   = calc_weight(add_op);
   ...
endfunction
```

Beware!! No longer truly random!

Copyright © 2009 by Doulos. All Rights Reserved 29

Notes

Before randomize() is called, a method called pre_randomized() is invoked. So we can write a pre_randomize() function that will update the weightings to be used in our dist constraint.

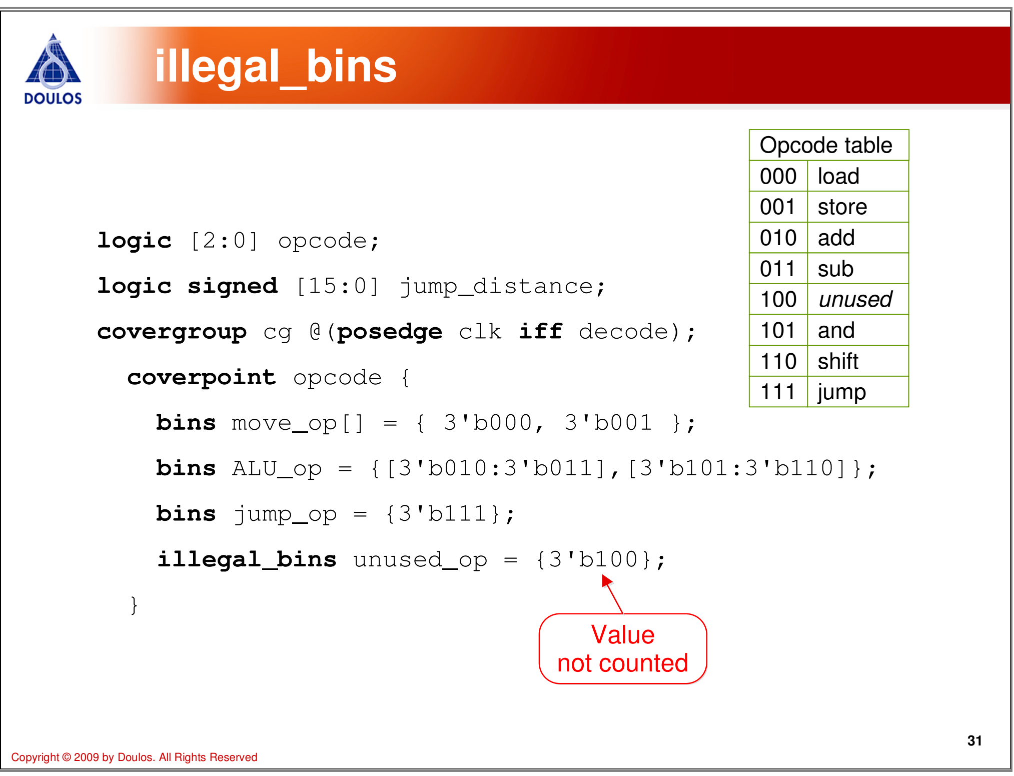
The calc_weight() function is called for each opcode and the coverage updated by grabbing the current coverage and subtracting it from 100. Using this formula, opcodes that have been seen a lot will have a small weight; whereas, unseen opcodes will have a very high probability of being selected next. The +1 is added for the scenario when an opcode has been 100% covered since a weighting of 0 would remove the possibility of that opcode from happening again. This way all opcodes continue to be selected.

The point of randomization is to find hard-to-find corner cases due to all the randomization. Just beware that when you constrain your randomization like this then it is no longer truly random, which may or may not be what you intended.

Copyright © 2009 by DOULOS. All Rights Reserved

Notes

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Illegal bins can be used to remove unused or illegal values from the overall coverage calculation.

Copyright © 2009 by DOULOS. All Rights Reserved

Issues

- **illegal_bins**
  - excludes values from a covergroup--that's good
  - throws errors--that's bad!

```
# ** Error: Illegal range bin value='b1011
got covered. The bin counter for the bin
'\/covunit/cg_i.b.bad' is 362.
```

- Questions to consider:
  - Should something passive throw errors?
  - If used for checking, what happens if coverage is turned off?
- Better option:
  - write assertions and checkers for checking
  - **ignore_bins** for coverage

32

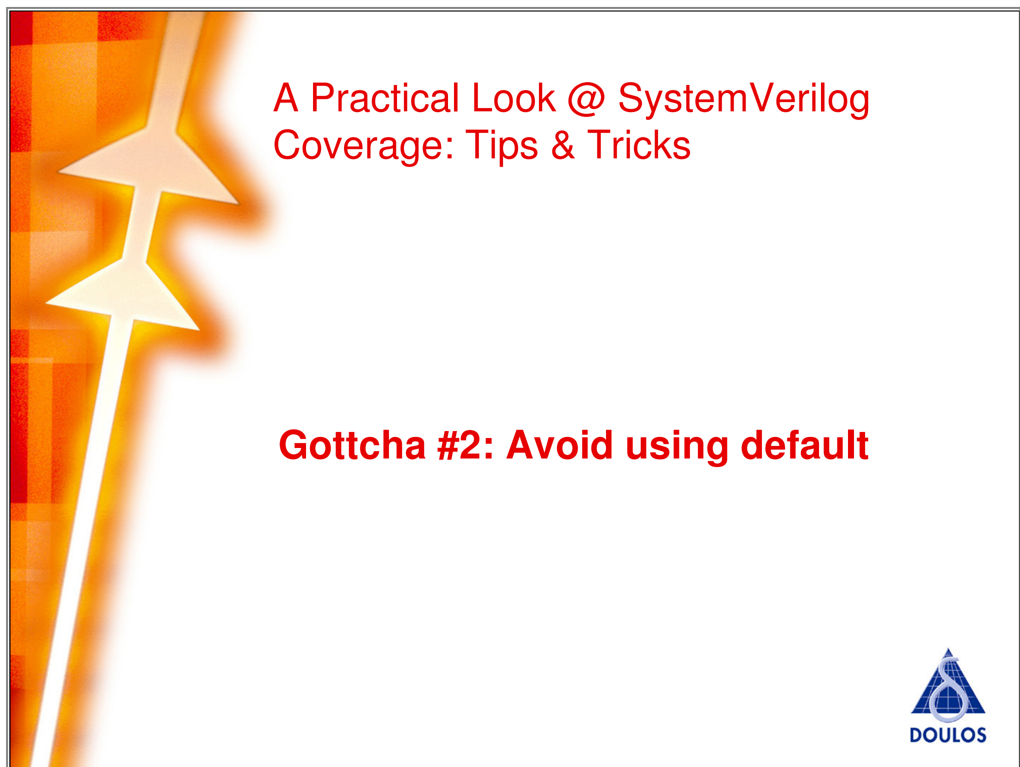
Copyright © 2009 by Doulos. All Rights Reserved

Notes

While illegal_bins removes values from coverage calculations, it also throws errors. Philosophically, you need to ask yourself the questions, (1) “Should a passive component like a covergroup be actively throwing errors?” and (2) “If you rely on the covergroup for checking, then what happens when you turn coverage off?”

If you really want to ignore values, then use ignore_bins. If you really want to throw errors, then use an assertion or checker!

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Copyright © 2009 by DOULOS. All Rights Reserved

Using default bins

```
bit [15:0] i;
```



```
covergroup cg_Short @(posedge Clock);
  coverpoint i {
    bins zero      = { 0 };
    bins tiny      = { [1:100] };
    bins hunds[3] = { 200,300,400,500,600,700,800,900 };
    bins huge      = { [1000:$] };
    ignore_bins ignore = { [501:599] };
    bins others[] = default;
  }
endgroup
```


Copyright © 2009 by Doulos. All Rights Reserved

34

Notes

The keyword default is used as a catch-all for all other possible values for a coverpoint that have not already been thrown into a bin. In the above example, the others[] = default will create a bin for every value not specified by the bins statements.

Copyright © 2009 by DOULOS. All Rights Reserved

Issues (1)

```
int a;                 // 2^32 values
covergroup cg ...;
   coverpoint a { bins other[] = default; }
endgroup
```

One bin for each value

- Use of **default** may crash your simulator:

```
# ** Fatal: The number of singleton values
exceeded the system limit of 2147483647 for
unconstrained array bin 'other' in Coverpoint
'a' of Covergroup instance '\/covunit/cg_i'.
```

- Do you really want to look at 2147483647 bins?

Copyright © 2009 by Doulos. All Rights Reserved

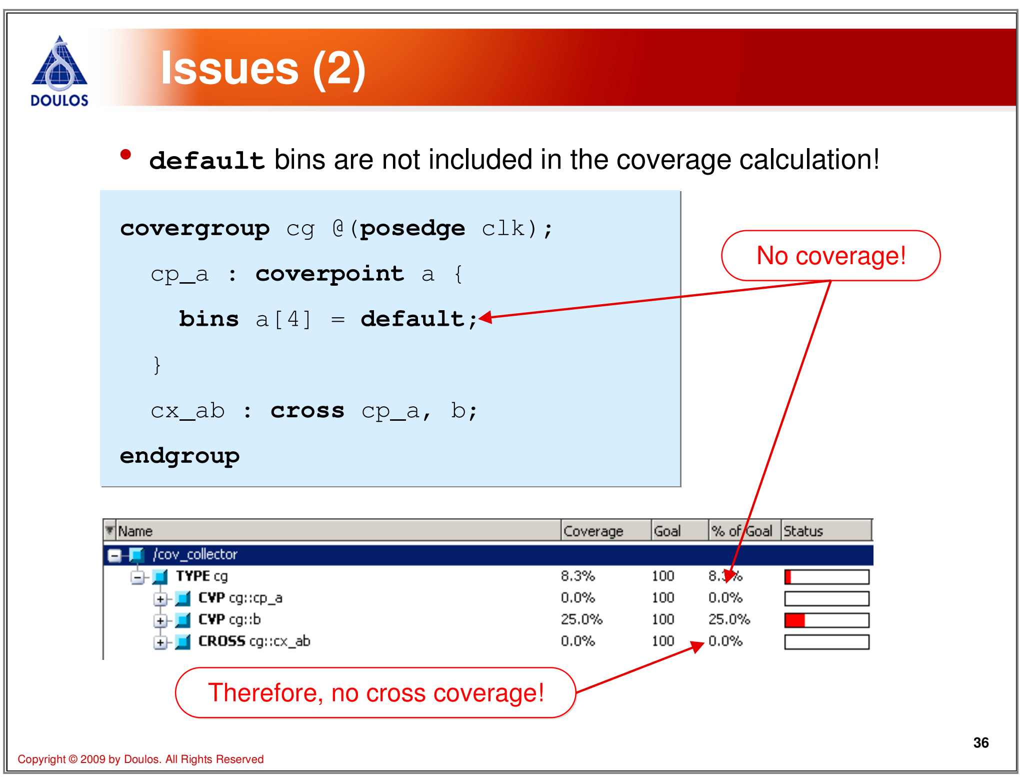
35

Notes

At first glance, default would appear quite useful. However, there are 2 issues. First, what if the coverpoint has a very large number of values? Some simulators croak on the example above!

It also begs the question, do you really want to look at 2147483647 bins? Most likely this is not what you intended.

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Another issue with default is that it pulls those values out of the coverage calculation. For example, suppose I wanted a shorthand way of taking all possible values and dividing them into several bins. Then I want to cross those values with another coverpoint. The obvious way to do this would be to use the default statement as shown above.

The problem with this example is that the coverpoint cp_a will have no coverage collected for it because it is using “default”. If the coverpoint has no coverage, then my cross will have no coverage either!

Copyright © 2009 by DOULOS. All Rights Reserved

Solution

- Avoid **[]** with **default,** or use with smaller variables with fewer possible values

```
logic [7:0] a;                  // Fewer values
covergroup cg ...;
   coverpoint a {
      bins other = default;     // One bin
   }
endgroup
```

- Use **wildcard** or min/max ($) to catch remaining values

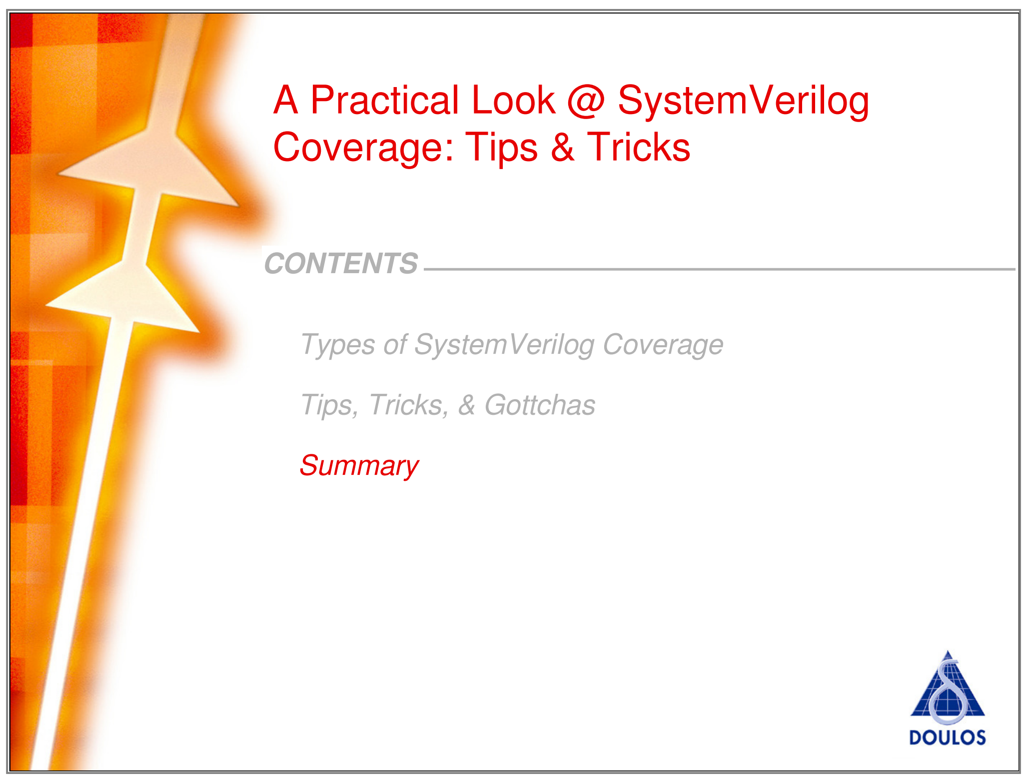
```
bins huge       = { [1000:$] };    // Max values
wildcard bins a[4] = { 'b?0 };     // Even values
```

Copyright © 2009 by Doulos. All Rights Reserved
37

Notes

The solutions to these issues is (1) do not use [ ] with the default statement or explicitly use ignore_bins, and (2) use $ or wildcard bins. The $ specifies min or max possible values and wildcard allows you to specify wildcard patterns. So if you want to capture all other possible values, you need to specify them using $ or wildcard.

Copyright © 2009 by DOULOS. All Rights Reserved



Notes

Copyright © 2009 by DOULOS. All Rights Reserved

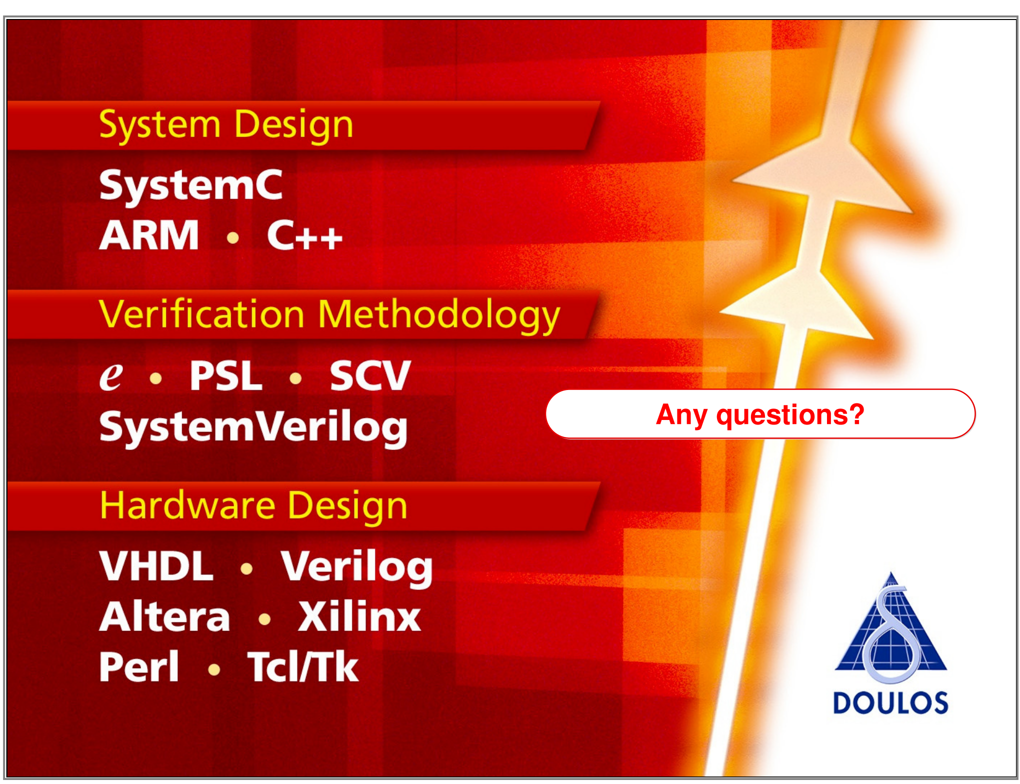
Summary

- Tip # 1: Take advantage of shorthand notation
- Tip # 2: Add covergroup arguments for more flexibility
- Tip # 3: Utilize coverage options
- Trick #1: Combine cover properties with covergroups
- Trick # 2: Create coverpoints for querying bin coverage
- Trick # 3: Direct stimulus with coverage feedback
- Gottcha # 1: Avoid illegal_bins
- Gottcha # 2: Avoid using default

Copyright © 2009 by Doulos. All Rights Reserved

39

Notes



Notes

Copyright © 2009 by DOULOS. All Rights Reserved